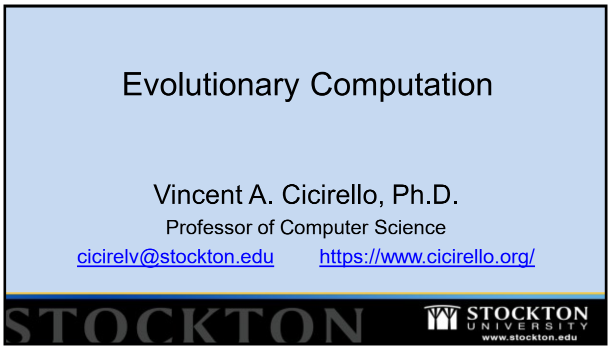
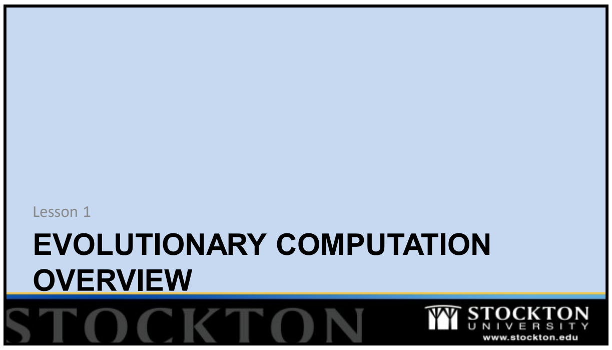


Evolutionary Computation

Vincent A. Cicirello, Ph.D.
Professor of Computer Science
cicirelv@stockton.edu https://www.cicirello.org/

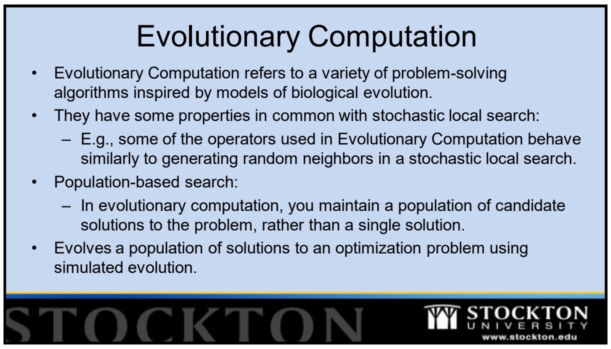
1



Lesson 1

EVOLUTIONARY COMPUTATION OVERVIEW

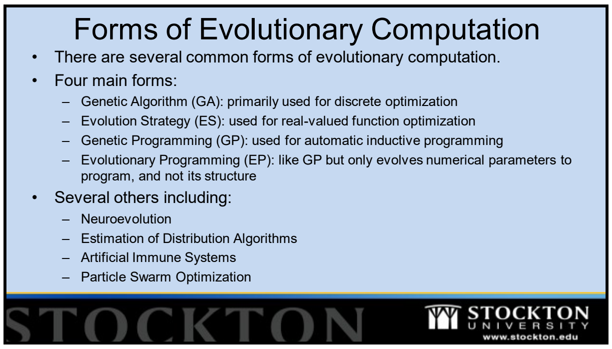
2



Evolutionary Computation

- Evolutionary Computation refers to a variety of problem-solving algorithms inspired by models of biological evolution.
- They have some properties in common with stochastic local search:
  - E.g., some of the operators used in Evolutionary Computation behave similarly to generating random neighbors in a stochastic local search.
- Population-based search:
  - In evolutionary computation, you maintain a population of candidate solutions to the problem, rather than a single solution.
- Evolves a population of solutions to an optimization problem using simulated evolution.

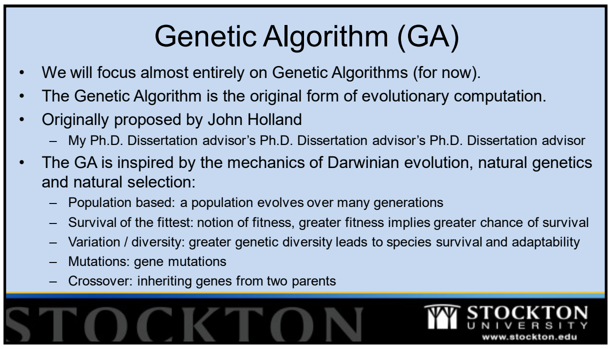
3



Forms of Evolutionary Computation

- There are several common forms of evolutionary computation.
- Four main forms:
  - Genetic Algorithm (GA): primarily used for discrete optimization
  - Evolution Strategy (ES): used for real-valued function optimization
  - Genetic Programming (GP): used for automatic inductive programming
  - Evolutionary Programming (EP): like GP but only evolves numerical parameters to program, and not its structure
- Several others including:
  - Neuroevolution
  - Estimation of Distribution Algorithms
  - Artificial Immune Systems
  - Particle Swarm Optimization

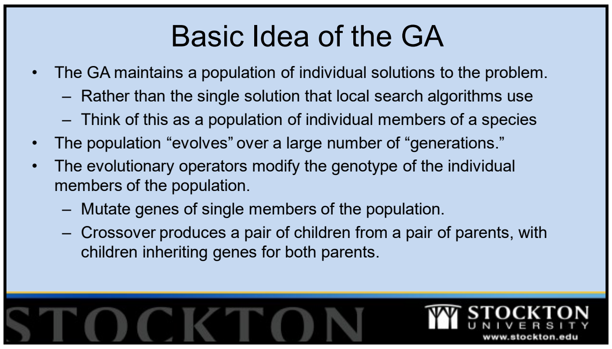
4



Genetic Algorithm (GA)

- We will focus almost entirely on Genetic Algorithms (for now).
- The Genetic Algorithm is the original form of evolutionary computation.
- Originally proposed by John Holland
  - My Ph.D. Dissertation advisor's Ph.D. Dissertation advisor's Ph.D. Dissertation advisor
- The GA is inspired by the mechanics of Darwinian evolution, natural genetics and natural selection:
  - Population based: a population evolves over many generations
  - Survival of the fittest: notion of fitness, greater fitness implies greater chance of survival
  - Variation / diversity: greater genetic diversity leads to species survival and adaptability
  - Mutations: gene mutations
  - Crossover: inheriting genes from two parents

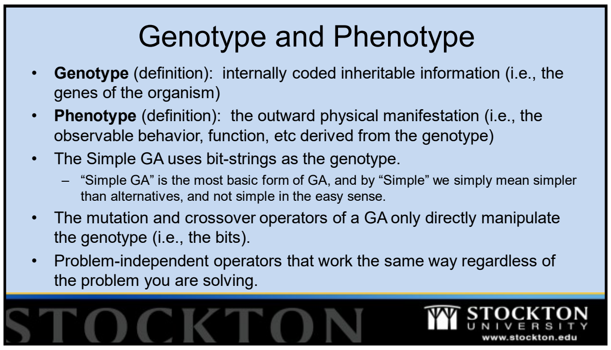
5



Basic Idea of the GA

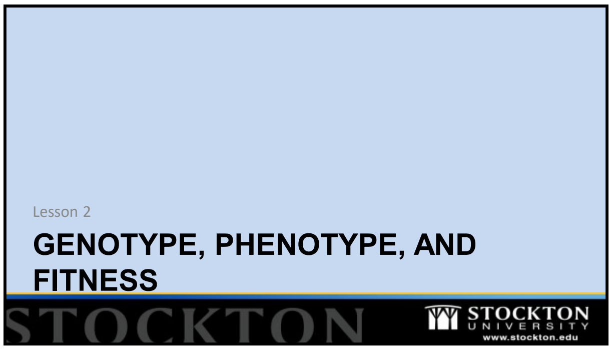
- The GA maintains a population of individual solutions to the problem.
  - Rather than the single solution that local search algorithms use
  - Think of this as a population of individual members of a species
- The population "evolves" over a large number of "generations."
- The evolutionary operators modify the genotype of the individual members of the population.
  - Mutate genes of single members of the population.
  - Crossover produces a pair of children from a pair of parents, with children inheriting genes for both parents.

6



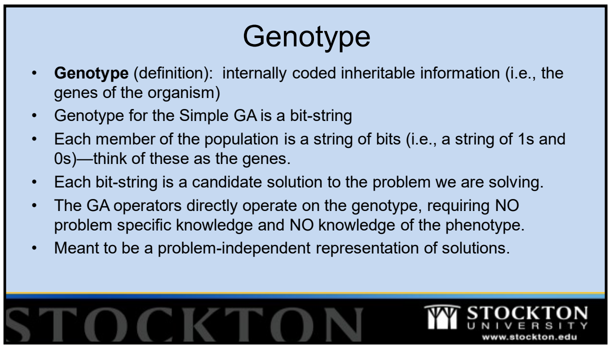
Genotype and Phenotype

- **Genotype** (definition): internally coded inheritable information (i.e., the genes of the organism)
- **Phenotype** (definition): the outward physical manifestation (i.e., the observable behavior, function, etc derived from the genotype)
- The Simple GA uses bit-strings as the genotype.
  - "Simple GA" is the most basic form of GA, and by "Simple" we simply mean simpler than alternatives, and not simple in the easy sense.
- The mutation and crossover operators of a GA only directly manipulate the genotype (i.e., the bits).
- Problem-independent operators that work the same way regardless of the problem you are solving.



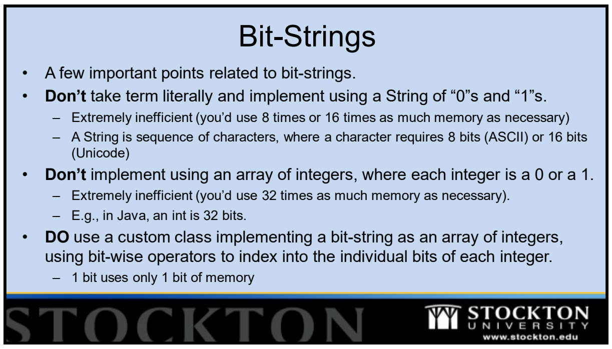
Lesson 2

GENOTYPE, PHENOTYPE, AND FITNESS



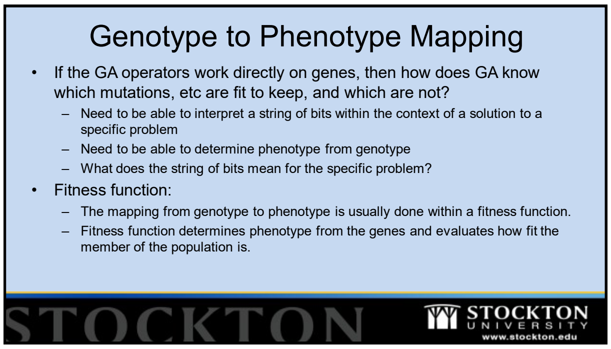
Genotype

- **Genotype** (definition): internally coded inheritable information (i.e., the genes of the organism)
- Genotype for the Simple GA is a bit-string
- Each member of the population is a string of bits (i.e., a string of 1s and 0s)—think of these as the genes.
- Each bit-string is a candidate solution to the problem we are solving.
- The GA operators directly operate on the genotype, requiring NO problem specific knowledge and NO knowledge of the phenotype.
- Meant to be a problem-independent representation of solutions.



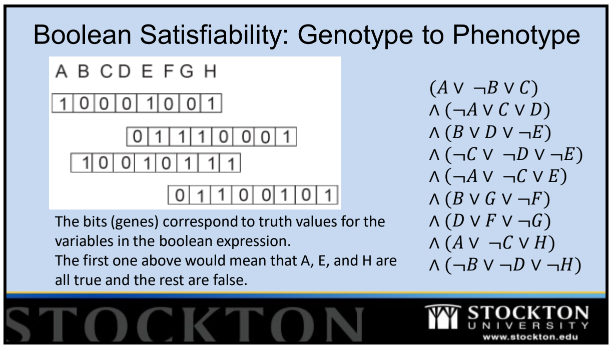
Bit-Strings

- A few important points related to bit-strings.
- **Don't** take term literally and implement using a String of "0"s and "1"s.
  - Extremely inefficient (you'd use 8 times or 16 times as much memory as necessary)
  - A String is sequence of characters, where a character requires 8 bits (ASCII) or 16 bits (Unicode)
- **Don't** implement using an array of integers, where each integer is a 0 or a 1.
  - Extremely inefficient (you'd use 32 times as much memory as necessary).
  - E.g., in Java, an int is 32 bits.
- **DO** use a custom class implementing a bit-string as an array of integers, using bit-wise operators to index into the individual bits of each integer.
  - 1 bit uses only 1 bit of memory

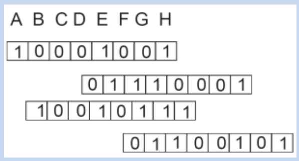


Genotype to Phenotype Mapping

- If the GA operators work directly on genes, then how does GA know which mutations, etc are fit to keep, and which are not?
  - Need to be able to interpret a string of bits within the context of a solution to a specific problem
  - Need to be able to determine phenotype from genotype
  - What does the string of bits mean for the specific problem?
- Fitness function:
  - The mapping from genotype to phenotype is usually done within a fitness function.
  - Fitness function determines phenotype from the genes and evaluates how fit the member of the population is.



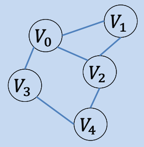
Boolean Satisfiability: Genotype to Phenotype



The bits (genes) correspond to truth values for the variables in the boolean expression.
The first one above would mean that A, E, and H are all true and the rest are false.

$$
\begin{aligned}
&(A \lor \neg B \lor C) \\
&\land (\neg A \lor C \lor D) \\
&\land (B \lor D \lor \neg E) \\
&\land (\neg C \lor \neg D \lor \neg E) \\
&\land (\neg A \lor \neg C \lor E) \\
&\land (B \lor G \lor \neg F) \\
&\land (D \lor F \lor \neg G) \\
&\land (A \lor \neg C \lor H) \\
&\land (\neg B \lor \neg D \lor \neg H)
\end{aligned}
$$

Graph Coloring: Genotype to Phenotype



Genotype: 0101110010

- Let each bit-string have length 2N, where N is number of graph vertexes.
- Every 2 bits encodes color of one vertex:
  - 00 = red
  - 01 = green
  - 10 = blue
  - 11 = red [Important: Every bit-string must map to a phenotype.]

Genotype: 01|01|11|00|10
Phenotype: V_0 V_1 V_2 V_3 V_4

STOCKTON UNIVERSITY www.stockton.edu

13

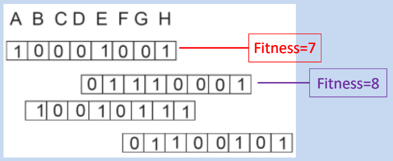
Fitness

- **Fitness** (definition): a numerical measure of the quality of a member of the population.
  - Depends on the phenotype
  - Measures how well the population member solves our problem
  - Relates to the criteria we are optimizing
- We need higher Fitness to imply better solution
  - May need to adjust the optimization criteria for problems where we are minimizing (e.g., for TSP, might use: 1/(tour length))
- Examples:
  - Boolean satisfiability: number of satisifed clauses
  - TSP: c / (tour length), where c is a constant
  - Graph Coloring: c / (1 + conflicts), where c is a constant

STOCKTON UNIVERSITY www.stockton.edu

14

Fitness Example



$$\begin{aligned}&(A \lor \neg B \lor C)\\ \land\ &(\neg A \lor C \lor D)\\ \land\ &(B \lor D \lor \neg E)\\ \land\ &(\neg C \lor \neg D \lor \neg E)\\ \land\ &(\neg A \lor \neg C \lor E)\\ \land\ &(B \lor G \lor \neg F)\\ \land\ &(D \lor F \lor \neg G)\\ \land\ &(A \lor \neg C \lor H)\\ \land\ &(\neg B \lor \neg D \lor \neg H)\end{aligned}$$

STOCKTON UNIVERSITY www.stockton.edu

15

Bit-strings and the TSP

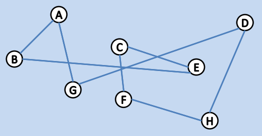
- How can we represent a tour of cities for a traveling salesperson problem with a string of bits?
- Genotype: Use a bit-string of length: $\lceil \lg N \rceil (N-1)$ for N-city instance.
- Mapping genotype to phenotype (i.e., bit-string to tour of cities):
  - Put list of cities in a circular list in some consistent order (e.g., by order of city ids).
  - Each group of $\lceil \lg N \rceil$ bits of the bit-string is an index into that circular list (note: an index into a circular list just wraps around if greater than length).
  - Use first $\lceil \lg N \rceil$ bits to select first city for tour, add it to start of tour, remove from circular list.
  - Use next $\lceil \lg N \rceil$ bits to select next city, add it to end of tour remove from circular list
  - Repeat until you have a tour.

STOCKTON UNIVERSITY www.stockton.edu

16

Bit-String to TSP Tour Example

- Bit-string must be length: $\lceil \lg N \rceil (N-1) = 21$
- A bit string in our GA population: 010 011 111 101 110 000 001
- Circular list initialized to:
  - A, B, C, D, E, F, G, H
- List: A, B, D, E, F, G, H; Tour: C
- List: A, B, D, F, G, H; Tour: C, E
- List: A, D, F, G, H; Tour: C, E, B
- List: D, F, G, H; Tour: C, E, B, A
- List: D, F, H; Tour: C, E, B, A, G
- List: F, H; Tour: C, E, B, A, G, D
- Tour: C, E, B, A, G, D, H, F



Consider this 8-city TSP as an example.

STOCKTON UNIVERSITY www.stockton.edu

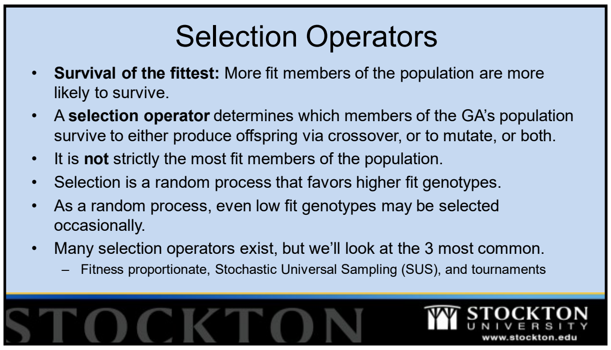
17

Lesson 3

SELECTION OPERATORS

STOCKTON UNIVERSITY www.stockton.edu

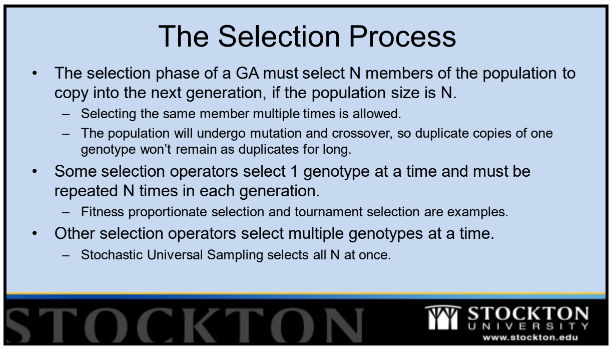
18



Selection Operators

- **Survival of the fittest:** More fit members of the population are more likely to survive.
- A **selection operator** determines which members of the GA's population survive to either produce offspring via crossover, or to mutate, or both.
- It is **not** strictly the most fit members of the population.
- Selection is a random process that favors higher fit genotypes.
- As a random process, even low fit genotypes may be selected occasionally.
- Many selection operators exist, but we'll look at the 3 most common.
  - Fitness proportionate, Stochastic Universal Sampling (SUS), and tournaments

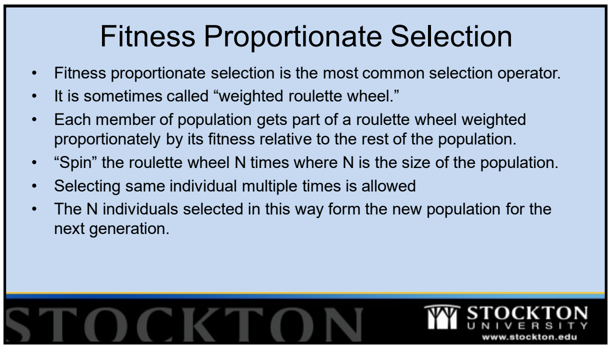
19



The Selection Process

- The selection phase of a GA must select N members of the population to copy into the next generation, if the population size is N.
  - Selecting the same member multiple times is allowed.
  - The population will undergo mutation and crossover, so duplicate copies of one genotype won't remain as duplicates for long.
- Some selection operators select 1 genotype at a time and must be repeated N times in each generation.
  - Fitness proportionate selection and tournament selection are examples.
- Other selection operators select multiple genotypes at a time.
  - Stochastic Universal Sampling selects all N at once.

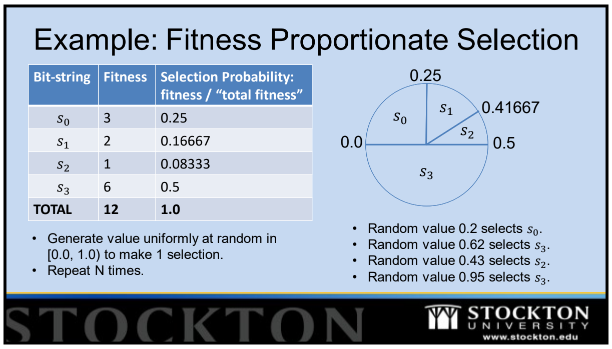
20



Fitness Proportionate Selection

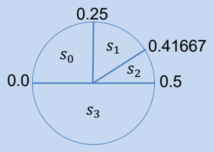
- Fitness proportionate selection is the most common selection operator.
- It is sometimes called "weighted roulette wheel."
- Each member of population gets part of a roulette wheel weighted proportionately by its fitness relative to the rest of the population.
- "Spin" the roulette wheel N times where N is the size of the population.
- Selecting same individual multiple times is allowed
- The N individuals selected in this way form the new population for the next generation.

21



Example: Fitness Proportionate Selection

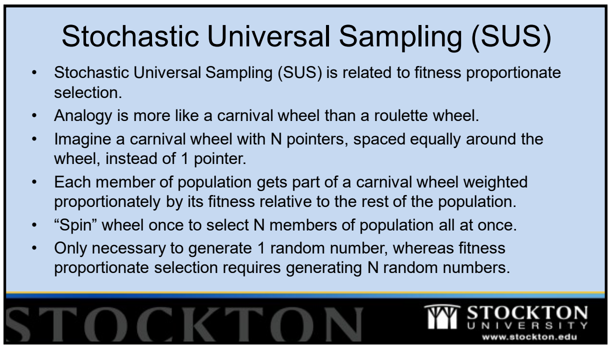
Bit-string	Fitness	Selection Probability: fitness / "total fitness"
s_0	3	0.25
s_1	2	0.16667
s_2	1	0.08333
s_3	6	0.5
TOTAL	12	1.0



- Generate value uniformly at random in [0.0, 1.0) to make 1 selection.
- Repeat N times.

- Random value 0.2 selects s_0.
- Random value 0.62 selects s_3.
- Random value 0.43 selects s_2.
- Random value 0.95 selects s_3.

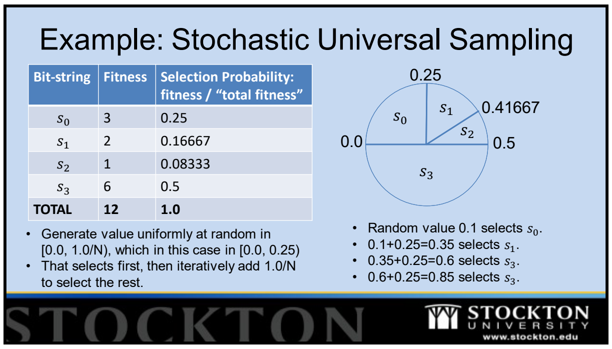
22



Stochastic Universal Sampling (SUS)

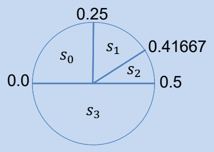
- Stochastic Universal Sampling (SUS) is related to fitness proportionate selection.
- Analogy is more like a carnival wheel than a roulette wheel.
- Imagine a carnival wheel with N pointers, spaced equally around the wheel, instead of 1 pointer.
- Each member of population gets part of a carnival wheel weighted proportionately by its fitness relative to the rest of the population.
- "Spin" wheel once to select N members of population all at once.
- Only necessary to generate 1 random number, whereas fitness proportionate selection requires generating N random numbers.

23



Example: Stochastic Universal Sampling

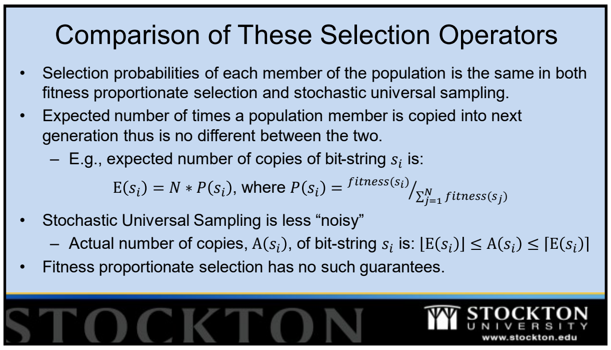
Bit-string	Fitness	Selection Probability: fitness / "total fitness"
s_0	3	0.25
s_1	2	0.16667
s_2	1	0.08333
s_3	6	0.5
TOTAL	12	1.0



- Generate value uniformly at random in [0.0, 1.0/N), which in this case in [0.0, 0.25)
- That selects first, then iteratively add 1.0/N to select the rest.

- Random value 0.1 selects s_0.
- 0.1+0.25=0.35 selects s_1.
- 0.35+0.25=0.6 selects s_3.
- 0.6+0.25=0.85 selects s_3.

24



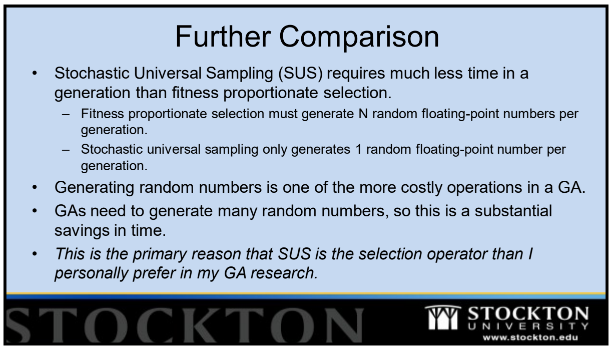
Comparison of These Selection Operators

- Selection probabilities of each member of the population is the same in both fitness proportionate selection and stochastic universal sampling.
- Expected number of times a population member is copied into next generation thus is no different between the two.
  - E.g., expected number of copies of bit-string s_i is:

$$\mathrm{E}(s_i) = N * P(s_i), \text{ where } P(s_i) = {fitness(s_i)}/{\sum_{j=1}^{N} fitness(s_j)}$$

- Stochastic Universal Sampling is less "noisy"
  - Actual number of copies, $\mathrm{A}(s_i)$, of bit-string s_i is: $\lfloor \mathrm{E}(s_i) \rfloor \leq \mathrm{A}(s_i) \leq \lceil \mathrm{E}(s_i) \rceil$
- Fitness proportionate selection has no such guarantees.

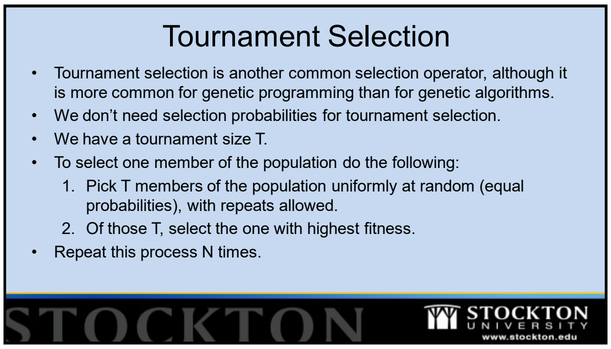
25



Further Comparison

- Stochastic Universal Sampling (SUS) requires much less time in a generation than fitness proportionate selection.
  - Fitness proportionate selection must generate N random floating-point numbers per generation.
  - Stochastic universal sampling only generates 1 random floating-point number per generation.
- Generating random numbers is one of the more costly operations in a GA.
- GAs need to generate many random numbers, so this is a substantial savings in time.
- *This is the primary reason that SUS is the selection operator than I personally prefer in my GA research.*

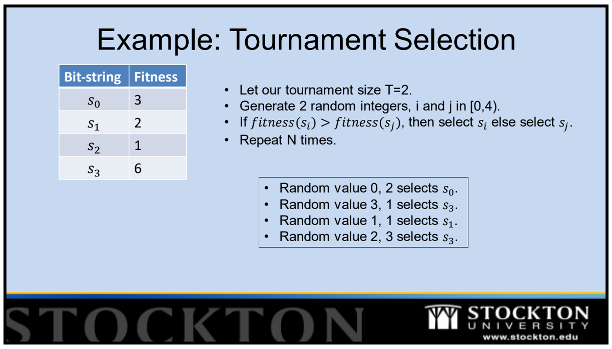
26



Tournament Selection

- Tournament selection is another common selection operator, although it is more common for genetic programming than for genetic algorithms.
- We don't need selection probabilities for tournament selection.
- We have a tournament size T.
- To select one member of the population do the following:
  1. Pick T members of the population uniformly at random (equal probabilities), with repeats allowed.
  2. Of those T, select the one with highest fitness.
- Repeat this process N times.

27



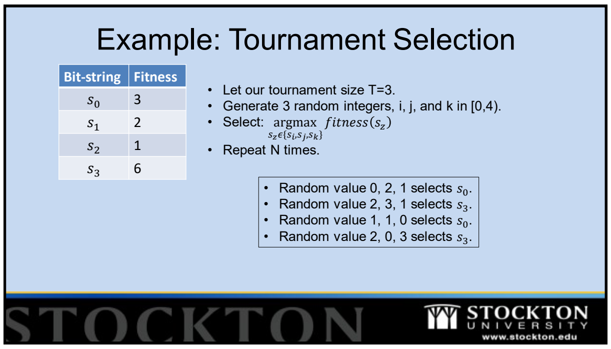
Example: Tournament Selection

Bit-string	Fitness
s_0	3
s_1	2
s_2	1
s_3	6

- Let our tournament size T=2.
- Generate 2 random integers, i and j in [0,4).
- If $fitness(s_i) > fitness(s_j)$, then select s_i else select s_j.
- Repeat N times.

- Random value 0, 2 selects s_0.
- Random value 3, 1 selects s_3.
- Random value 1, 1 selects s_1.
- Random value 2, 3 selects s_3.

28



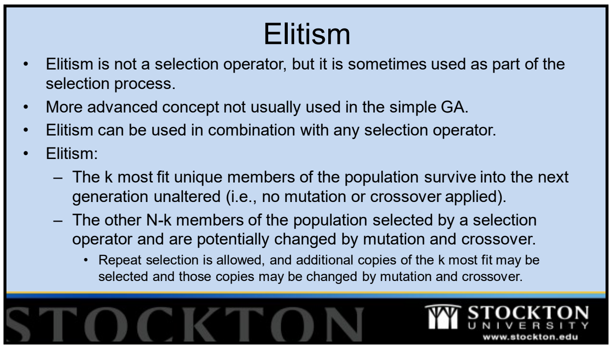
Example: Tournament Selection

Bit-string	Fitness
s_0	3
s_1	2
s_2	1
s_3	6

- Let our tournament size T=3.
- Generate 3 random integers, i, j, and k in [0,4).
- Select: $\underset{s_z \in \{s_i, s_j, s_k\}}{\operatorname{argmax}} \; fitness(s_z)$
- Repeat N times.

- Random value 0, 2, 1 selects s_0.
- Random value 2, 3, 1 selects s_3.
- Random value 1, 1, 0 selects s_0.
- Random value 2, 0, 3 selects s_3.

29



Elitism

- Elitism is not a selection operator, but it is sometimes used as part of the selection process.
- More advanced concept not usually used in the simple GA.
- Elitism can be used in combination with any selection operator.
- Elitism:
  - The k most fit unique members of the population survive into the next generation unaltered (i.e., no mutation or crossover applied).
  - The other N-k members of the population selected by a selection operator and are potentially changed by mutation and crossover.
    - Repeat selection is allowed, and additional copies of the k most fit may be selected and those copies may be changed by mutation and crossover.

30

Survival of the Fittest

- The selection operator produces the effect of survival of the fittest.
- **Question:** What happens if we begin with a random initial population of bit-strings, and then iteratively repeat Selection large number of times?
  - Without using any other operators (no mutation and no crossover)...
- **Answer:**
  - With high probability, we eventually get a population full of copies of the best individual from the initial population.
  - Selection by itself is a “noisy” and expensive way of selecting the best from a set (not particularly useful by itself).
- We need to combine selection with other operators.

STOCKTON UNIVERSITY www.stockton.edu

31

Lesson 4

MUTATION

STOCKTON UNIVERSITY www.stockton.edu

32

Mutation

- **Mutation:** A small random change to the genotype of a member of the GA population.
- Mutation is similar to the neighborhood function of a local search, especially in how random neighbors are chosen in stochastic hill climbing or in simulated annealing.

STOCKTON UNIVERSITY www.stockton.edu

33

Bit Flip Mutation

- Bit flip mutation is the most commonly used mutation operator for a GA.
  - When we look at alternatives to the bit-string, we'll see others.
- Let M be the **mutation rate**, such that $0.0 < M < 1.0$.
- The mutation rate is the probability that any given bit will be flipped in a given generation.
  - Flipping a bit means if it is a 0, change to a 1 (and vice versa).
- Procedure:

```
for each bit-string $s_j$ in the population
    for each bit $b_i$ of $s_j$
        Let r be uniformly random from [0.0, 1.0)
        if r < M then flip $b_i$
```

STOCKTON UNIVERSITY www.stockton.edu

34

Mutation Example

Let M=0.1 in this example.

Before Mutation:	0	1	1	1	0	0	1	0	0	1
Generate random numbers:	0.2	0.33	0.11	0.07	0.92	0.56	0.73	0.01	0.62	0.3
After Mutation:	0	1	1	0	0	0	1	1	0	1

STOCKTON UNIVERSITY www.stockton.edu

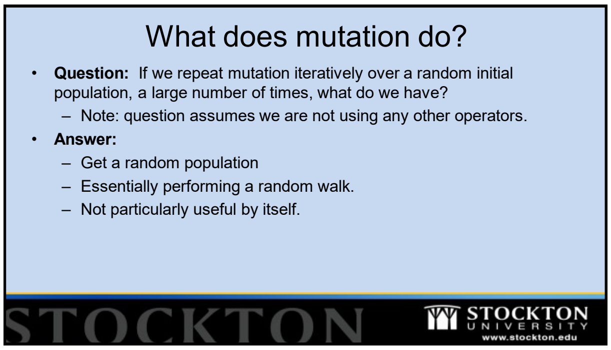
35

Setting the Mutation Rate

- How do you choose the value of the mutation rate M?
- Only technical requirement: $0.0 < M < 1.0$.
  - 0.0 would mean no mutation.
  - 1.0 would mean in every generation flip all bits, which would just oscillate the population back and forth.
- M is usually very low
  - Mutation is intended to make small changes
  - Similar to neighborhood function for local search
- Just how low depends on length of bit-strings
  - Common to set $M = {}^{1}/_{L}$, where L is length of a bit-string.
  - Or to set $M = {}^{c}/_{L}$, where c is a small constant, the average number of bits to flip.

STOCKTON UNIVERSITY www.stockton.edu

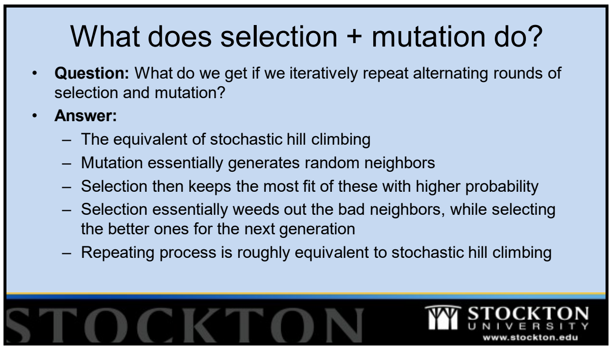
36



What does mutation do?

- **Question:** If we repeat mutation iteratively over a random initial population, a large number of times, what do we have?
  - Note: question assumes we are not using any other operators.
- **Answer:**
  - Get a random population
  - Essentially performing a random walk.
  - Not particularly useful by itself.

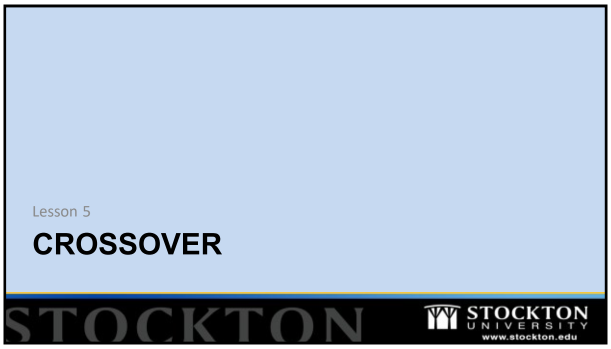
37



What does selection + mutation do?

- **Question:** What do we get if we iteratively repeat alternating rounds of selection and mutation?
- **Answer:**
  - The equivalent of stochastic hill climbing
  - Mutation essentially generates random neighbors
  - Selection then keeps the most fit of these with higher probability
  - Selection essentially weeds out the bad neighbors, while selecting the better ones for the next generation
  - Repeating process is roughly equivalent to stochastic hill climbing

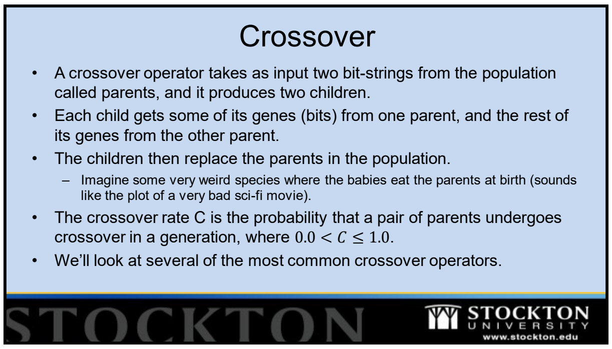
38



Lesson 5

CROSSOVER

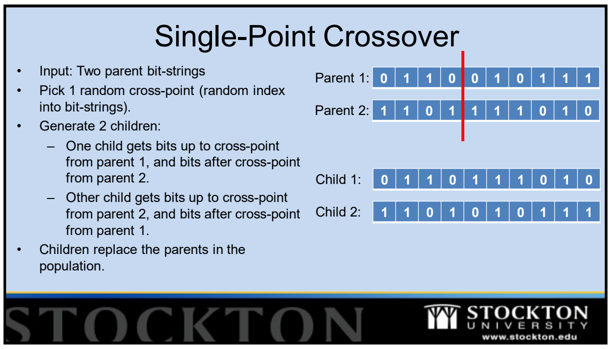
39



Crossover

- A crossover operator takes as input two bit-strings from the population called parents, and it produces two children.
- Each child gets some of its genes (bits) from one parent, and the rest of its genes from the other parent.
- The children then replace the parents in the population.
  - Imagine some very weird species where the babies eat the parents at birth (sounds like the plot of a very bad sci-fi movie).
- The crossover rate C is the probability that a pair of parents undergoes crossover in a generation, where $0.0 < C \leq 1.0$.
- We'll look at several of the most common crossover operators.

40

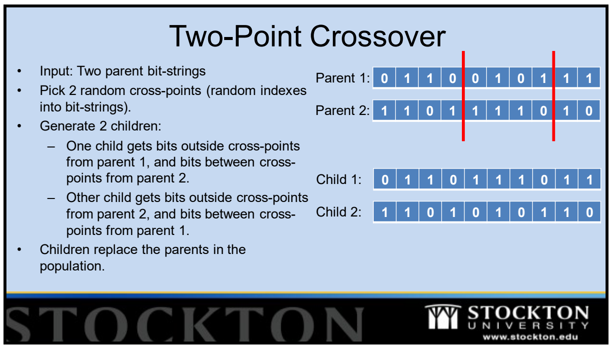


Single-Point Crossover

- Input: Two parent bit-strings
- Pick 1 random cross-point (random index into bit-strings).
- Generate 2 children:
  - One child gets bits up to cross-point from parent 1, and bits after cross-point from parent 2.
  - Other child gets bits up to cross-point from parent 2, and bits after cross-point from parent 1.
- Children replace the parents in the population.

Parent 1:	0	1	1	0	0	1	0	1	1	1
Parent 2:	1	1	0	1	1	1	1	0	1	0
Child 1:	0	1	1	0	1	1	1	0	1	0
Child 2:	1	1	0	1	0	1	0	1	1	1

41

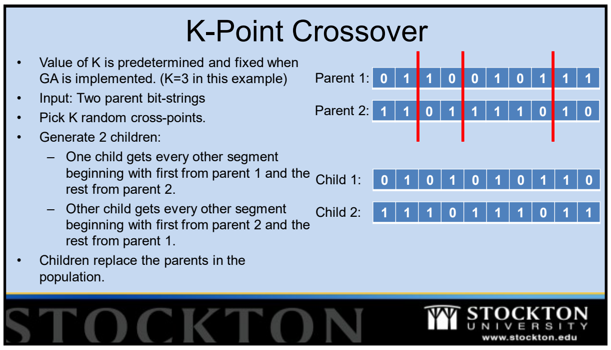


Two-Point Crossover

- Input: Two parent bit-strings
- Pick 2 random cross-points (random indexes into bit-strings).
- Generate 2 children:
  - One child gets bits outside cross-points from parent 1, and bits between cross-points from parent 2.
  - Other child gets bits outside cross-points from parent 2, and bits between cross-points from parent 1.
- Children replace the parents in the population.

Parent 1:	0	1	1	0	0	1	0	1	1	1
Parent 2:	1	1	0	1	1	1	1	0	1	0
Child 1:	0	1	1	0	1	1	1	0	1	1
Child 2:	1	1	0	1	0	1	0	1	1	0

42

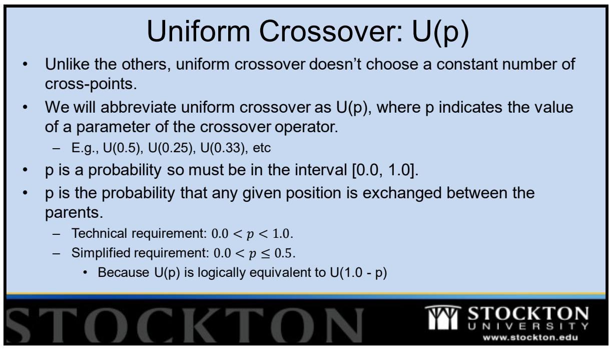


K-Point Crossover

- Value of K is predetermined and fixed when GA is implemented. (K=3 in this example)
- Input: Two parent bit-strings
- Pick K random cross-points.
- Generate 2 children:
  - One child gets every other segment beginning with first from parent 1 and the rest from parent 2.
  - Other child gets every other segment beginning with first from parent 2 and the rest from parent 1.
- Children replace the parents in the population.

Parent 1:	0	1	1	0	0	1	0	1	1	1
Parent 2:	1	1	0	1	1	1	1	0	1	0
Child 1:	0	1	0	1	0	1	0	1	1	0
Child 2:	1	1	1	0	1	1	1	0	1	1

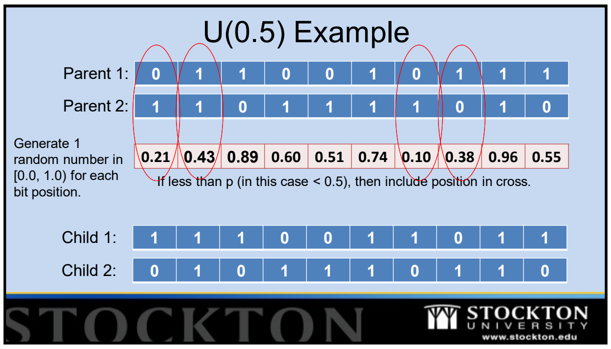
43



Uniform Crossover: U(p)

- Unlike the others, uniform crossover doesn't choose a constant number of cross-points.
- We will abbreviate uniform crossover as U(p), where p indicates the value of a parameter of the crossover operator.
  - E.g., U(0.5), U(0.25), U(0.33), etc
- p is a probability so must be in the interval [0.0, 1.0].
- p is the probability that any given position is exchanged between the parents.
  - Technical requirement: $0.0 < p < 1.0$.
  - Simplified requirement: $0.0 < p \leq 0.5$.
    - Because U(p) is logically equivalent to U(1.0 - p)

44

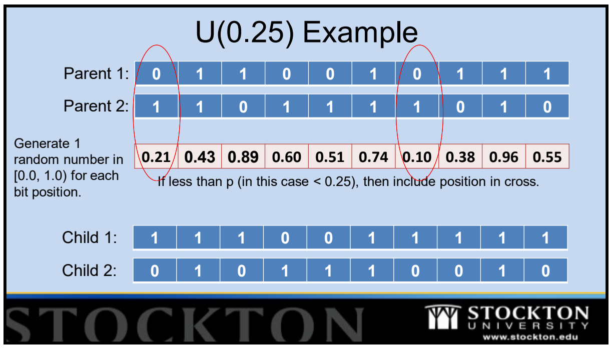


U(0.5) Example

Parent 1:	0	1	1	0	0	1	0	1	1	1
Parent 2:	1	1	0	1	1	1	1	0	1	0
Generate 1 random number in [0.0, 1.0) for each bit position.	0.21	0.43	0.89	0.60	0.51	0.74	0.10	0.38	0.96	0.55
Child 1:	1	1	1	0	0	1	1	0	1	1
Child 2:	0	1	0	1	1	1	0	1	1	0

If less than p (in this case < 0.5), then include position in cross.

45

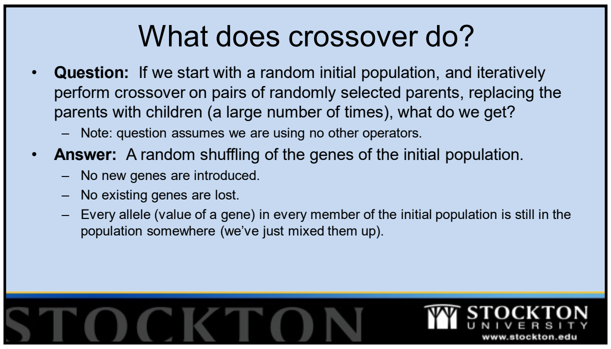


U(0.25) Example

Parent 1:	0	1	1	0	0	1	0	1	1	1
Parent 2:	1	1	0	1	1	1	1	0	1	0
Generate 1 random number in [0.0, 1.0) for each bit position.	0.21	0.43	0.89	0.60	0.51	0.74	0.10	0.38	0.96	0.55
Child 1:	1	1	1	0	0	1	1	1	1	1
Child 2:	0	1	0	1	1	1	0	0	1	0

If less than p (in this case < 0.25), then include position in cross.

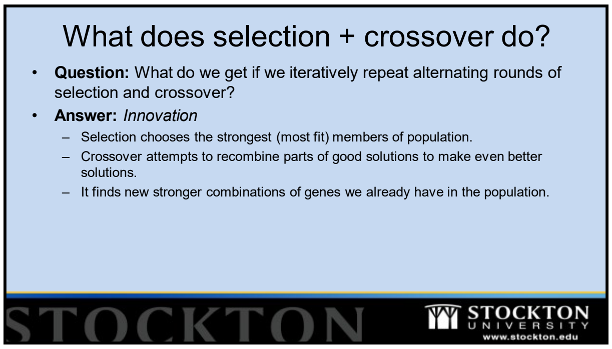
46



What does crossover do?

- **Question:** If we start with a random initial population, and iteratively perform crossover on pairs of randomly selected parents, replacing the parents with children (a large number of times), what do we get?
  - Note: question assumes we are using no other operators.
- **Answer:** A random shuffling of the genes of the initial population.
  - No new genes are introduced.
  - No existing genes are lost.
  - Every allele (value of a gene) in every member of the initial population is still in the population somewhere (we've just mixed them up).

47



What does selection + crossover do?

- **Question:** What do we get if we iteratively repeat alternating rounds of selection and crossover?
- **Answer:** *Innovation*
  - Selection chooses the strongest (most fit) members of population.
  - Crossover attempts to recombine parts of good solutions to make even better solutions.
  - It finds new stronger combinations of genes we already have in the population.

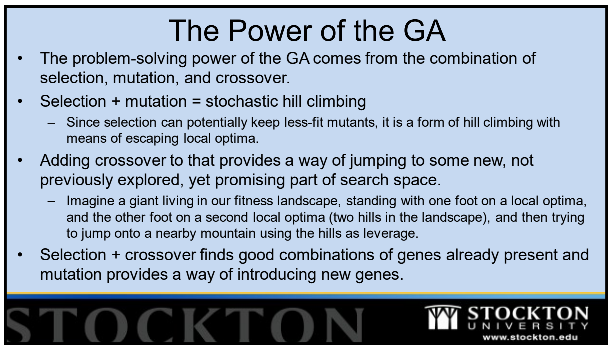
48

Lesson 6

THE GA: PUTTING IT ALL TOGETHER

STOCKTON UNIVERSITY www.stockton.edu

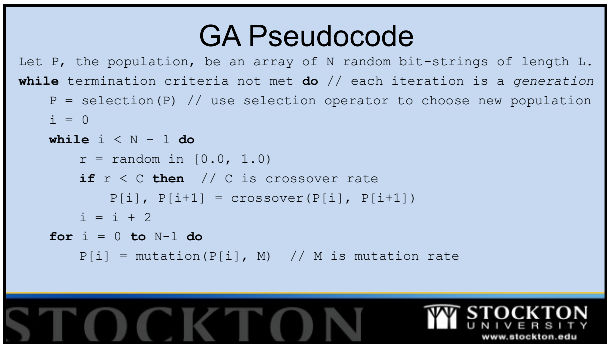
49


The Power of the GA

- The problem-solving power of the GA comes from the combination of selection, mutation, and crossover.
- Selection + mutation = stochastic hill climbing
  - Since selection can potentially keep less-fit mutants, it is a form of hill climbing with means of escaping local optima.
- Adding crossover to that provides a way of jumping to some new, not previously explored, yet promising part of search space.
  - Imagine a giant living in our fitness landscape, standing with one foot on a local optima, and the other foot on a second local optima (two hills in the landscape), and then trying to jump onto a nearby mountain using the hills as leverage.
- Selection + crossover finds good combinations of genes already present and mutation provides a way of introducing new genes.

STOCKTON UNIVERSITY www.stockton.edu

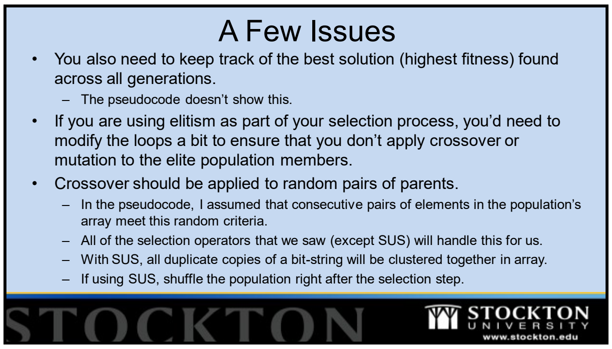
50


GA Pseudocode

```
Let P, the population, be an array of N random bit-strings of length L.
while termination criteria not met do // each iteration is a generation
    P = selection(P) // use selection operator to choose new population
    i = 0
    while i < N - 1 do
        r = random in [0.0, 1.0)
        if r < C then  // C is crossover rate
            P[i], P[i+1] = crossover(P[i], P[i+1])
        i = i + 2
    for i = 0 to N-1 do
        P[i] = mutation(P[i], M)  // M is mutation rate
```

STOCKTON UNIVERSITY www.stockton.edu

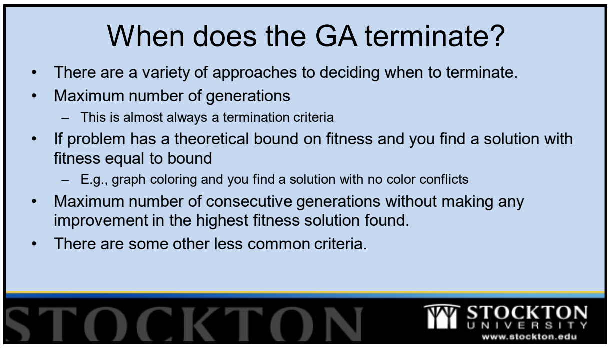
51


A Few Issues

- You also need to keep track of the best solution (highest fitness) found across all generations.
  - The pseudocode doesn't show this.
- If you are using elitism as part of your selection process, you'd need to modify the loops a bit to ensure that you don't apply crossover or mutation to the elite population members.
- Crossover should be applied to random pairs of parents.
  - In the pseudocode, I assumed that consecutive pairs of elements in the population's array meet this random criteria.
  - All of the selection operators that we saw (except SUS) will handle this for us.
  - With SUS, all duplicate copies of a bit-string will be clustered together in array.
  - If using SUS, shuffle the population right after the selection step.

STOCKTON UNIVERSITY www.stockton.edu

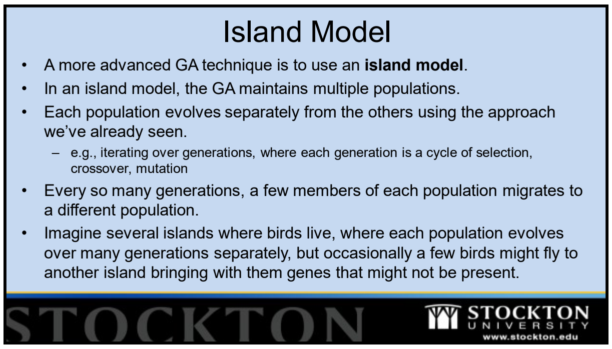
52


When does the GA terminate?

- There are a variety of approaches to deciding when to terminate.
- Maximum number of generations
  - This is almost always a termination criteria
- If problem has a theoretical bound on fitness and you find a solution with fitness equal to bound
  - E.g., graph coloring and you find a solution with no color conflicts
- Maximum number of consecutive generations without making any improvement in the highest fitness solution found.
- There are some other less common criteria.

STOCKTON UNIVERSITY www.stockton.edu

53


Island Model

- A more advanced GA technique is to use an **island model**.
- In an island model, the GA maintains multiple populations.
- Each population evolves separately from the others using the approach we've already seen.
  - e.g., iterating over generations, where each generation is a cycle of selection, crossover, mutation
- Every so many generations, a few members of each population migrates to a different population.
- Imagine several islands where birds live, where each population evolves over many generations separately, but occasionally a few birds might fly to another island bringing with them genes that might not be present.

STOCKTON UNIVERSITY www.stockton.edu

54

Parallel Genetic Algorithms

- Genetic Algorithms lend themselves nicely to parallel programming, enabling exploiting multi-core processors.
- There are a couple primary approaches to implementing a GA in parallel.
  1. Multiple populations: consider the island model but where each population evolves using a separate thread or process.
  2. One large population: multiple threads or processes share in the work of a generation (e.g., mutating different members of the population, applying crossover to different pairs of parents, etc)

55

Lesson 7

HOLLAND'S SCHEMA THEOREM

56

The Schema Theorem

- The **Schema Theorem** states that the number of short, low-order schemata with above-average fitness increases exponentially within the population.
- What does this mean? And why do we care?

57

What are Schemata?

- A **Schema** is a template or a pattern describing a set of bit-strings that share the same bit values at some positions.
- Some examples:
  - `101**1**0`
  - `**101****`
  - `1*0*11`
- The * are wildcards indicating any value (i.e., could be a 0 or a 1).
- A Schema defines a set of bit-strings.
  - E.g., the schema, `1*0*11` defines the set, {`100011, 100111, 110011, 110111`}
- Schemata is the plural of schema

58

The Order of a Schema

- The **order** of a schema, S, usually denoted o(S) is the number of fixed positions (non-wildcards) in the schema (i.e., positions with a 1 or a 0).
- What is the order of each of the following schemata?

S	o(S)
`101**1**0`	5
`**101****`	3
`1*0*1`	3
`**11*1*0*`	4

59

Defining Length of a Schema

- The **defining length** of a schema, S, usually denoted $\delta(S)$ is the length of the substring beginning at the first fixed position and ending at the last fixed position.
- What is the defining length of each of these schemata?

S	$\delta(S)$
`101**1**0`	9
`**101****`	3
`1*0*1`	5
`**11*1*0*`	6

60

Fitness of a Schema

- The fitness of a schema is the average fitness of all bit-strings that match the schema.
- E.g., the fitness of `1*0*11` is the average of the fitness of the following 4 schemata: `100011`, `100111`, `110011`, and `110111`.
- In this way, you can attribute fitness to parts of a bit-string.

61

Holland's Schema Theorem

- The **Schema Theorem** states that the number of short, low-order schemata with above-average fitness increases exponentially within the population.
- What does this mean?
  - Short: short defining length
  - Low-order: low number of non-wildcards
  - Above-average fitness: schemata with fitness above the average fitness of all possible schemata
- In other words, the number of small high-quality genetic building blocks in the population increases at an exponential rate.
- The GA has access to very good raw genetic material to work with.

62

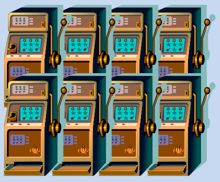
Holland's Schema Theorem

- The **Schema Theorem** states that the number of short, low-order schemata with above-average fitness increases exponentially within the population.
- But why do we care specifically about the exponential growth of above-average schemata?

63

The K-Armed Bandit Problem

- Given…
  - Slot machine with k arms
  - Different payoff distribution for each arm
  - We don't know what the specific payoff distributions are
- What's the optimal sampling policy to maximize expected sum of rewards?
- *Allocate trials to the observed best arm at a rate that increases exponentially* [Holland, '75].



64

Schema Theorem and the K-Armed Bandit

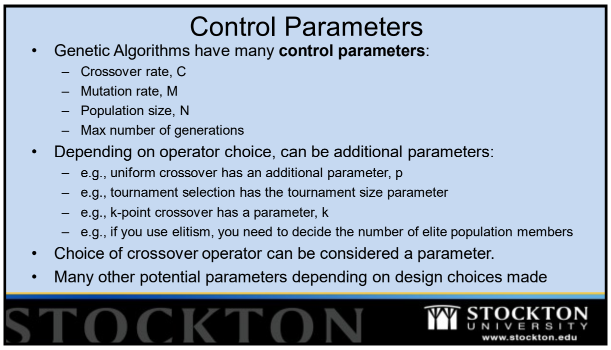
- The various schemata in the current population of the GA are like the k arms of the k-armed slot machine.
- Holland (in proof of schema theorem) showed that the number of short, low-order schemata with above-average fitness increases at an exponential rate during a run of a GA.
  - i.e., the GA is sampling the slot machine arms that appear to be best at an exponentially increasing rate
- Holland's analysis of the k-armed bandit shows that to maximize long term expected rewards, we need to allocate an exponentially increasing number of trials to the arm that we observe to be the best.
  - The GA produces the desired effect.

65

Lesson 8

CONTROL PARAMETERS

66

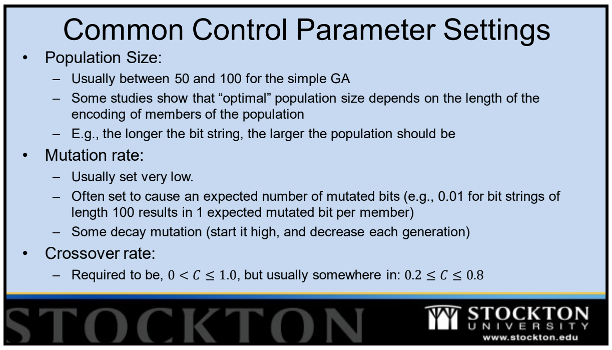


Control Parameters

- Genetic Algorithms have many **control parameters**:
  - Crossover rate, C
  - Mutation rate, M
  - Population size, N
  - Max number of generations
- Depending on operator choice, can be additional parameters:
  - e.g., uniform crossover has an additional parameter, p
  - e.g., tournament selection has the tournament size parameter
  - e.g., k-point crossover has a parameter, k
  - e.g., if you use elitism, you need to decide the number of elite population members
- Choice of crossover operator can be considered a parameter.
- Many other potential parameters depending on design choices made

STOCKTON UNIVERSITY www.stockton.edu

67

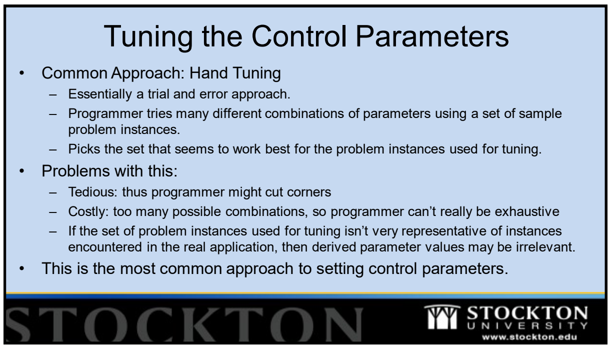


Common Control Parameter Settings

- Population Size:
  - Usually between 50 and 100 for the simple GA
  - Some studies show that "optimal" population size depends on the length of the encoding of members of the population
  - E.g., the longer the bit string, the larger the population should be
- Mutation rate:
  - Usually set very low.
  - Often set to cause an expected number of mutated bits (e.g., 0.01 for bit strings of length 100 results in 1 expected mutated bit per member)
  - Some decay mutation (start it high, and decrease each generation)
- Crossover rate:
  - Required to be, $0 < C \leq 1.0$, but usually somewhere in: $0.2 \leq C \leq 0.8$

STOCKTON UNIVERSITY www.stockton.edu

68

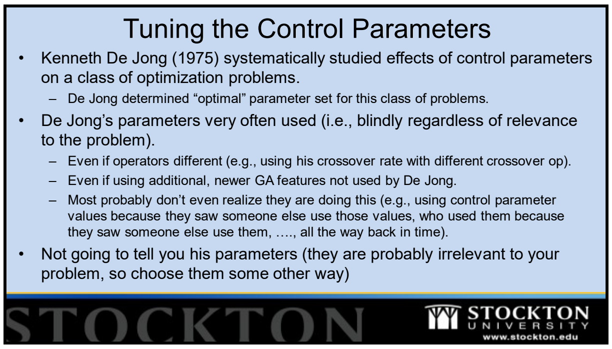


Tuning the Control Parameters

- Common Approach: Hand Tuning
  - Essentially a trial and error approach.
  - Programmer tries many different combinations of parameters using a set of sample problem instances.
  - Picks the set that seems to work best for the problem instances used for tuning.
- Problems with this:
  - Tedious: thus programmer might cut corners
  - Costly: too many possible combinations, so programmer can't really be exhaustive
  - If the set of problem instances used for tuning isn't very representative of instances encountered in the real application, then derived parameter values may be irrelevant.
- This is the most common approach to setting control parameters.

STOCKTON UNIVERSITY www.stockton.edu

69

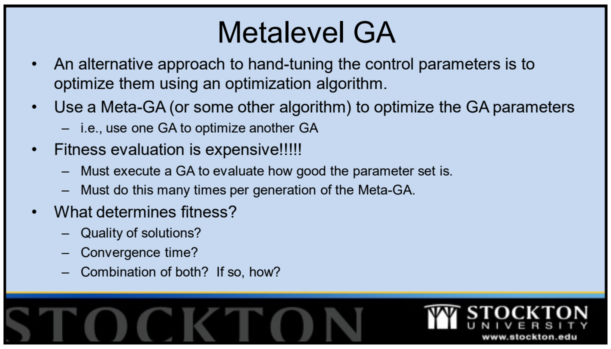


Tuning the Control Parameters

- Kenneth De Jong (1975) systematically studied effects of control parameters on a class of optimization problems.
  - De Jong determined "optimal" parameter set for this class of problems.
- De Jong's parameters very often used (i.e., blindly regardless of relevance to the problem).
  - Even if operators different (e.g., using his crossover rate with different crossover op).
  - Even if using additional, newer GA features not used by De Jong.
  - Most probably don't even realize they are doing this (e.g., using control parameter values because they saw someone else use those values, who used them because they saw someone else use them, …., all the way back in time).
- Not going to tell you his parameters (they are probably irrelevant to your problem, so choose them some other way)

STOCKTON UNIVERSITY www.stockton.edu

70

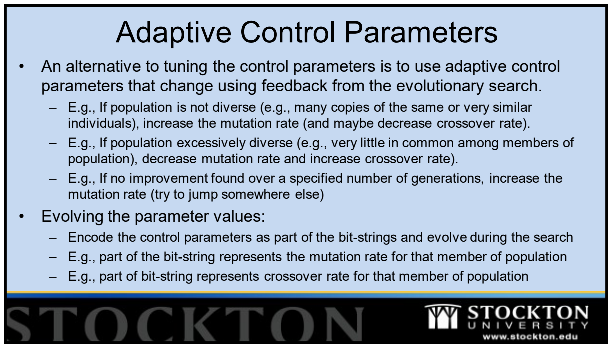


Metalevel GA

- An alternative approach to hand-tuning the control parameters is to optimize them using an optimization algorithm.
- Use a Meta-GA (or some other algorithm) to optimize the GA parameters
  - i.e., use one GA to optimize another GA
- Fitness evaluation is expensive!!!!!
  - Must execute a GA to evaluate how good the parameter set is.
  - Must do this many times per generation of the Meta-GA.
- What determines fitness?
  - Quality of solutions?
  - Convergence time?
  - Combination of both? If so, how?

STOCKTON UNIVERSITY www.stockton.edu

71

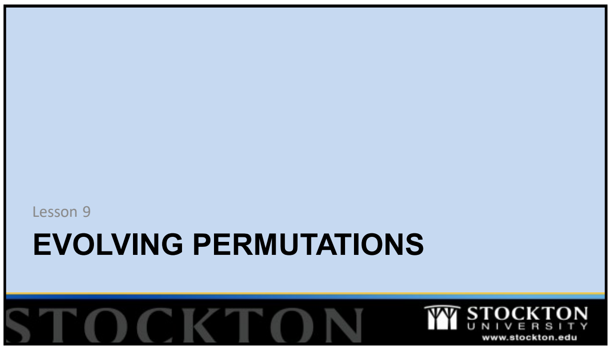


Adaptive Control Parameters

- An alternative to tuning the control parameters is to use adaptive control parameters that change using feedback from the evolutionary search.
  - E.g., If population is not diverse (e.g., many copies of the same or very similar individuals), increase the mutation rate (and maybe decrease crossover rate).
  - E.g., If population excessively diverse (e.g., very little in common among members of population), decrease mutation rate and increase crossover rate).
  - E.g., If no improvement found over a specified number of generations, increase the mutation rate (try to jump somewhere else)
- Evolving the parameter values:
  - Encode the control parameters as part of the bit-strings and evolve during the search
  - E.g., part of the bit-string represents the mutation rate for that member of population
  - E.g., part of bit-string represents crossover rate for that member of population

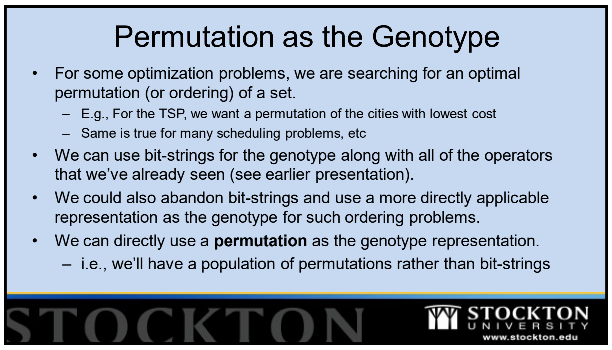
STOCKTON UNIVERSITY www.stockton.edu

72


Lesson 9

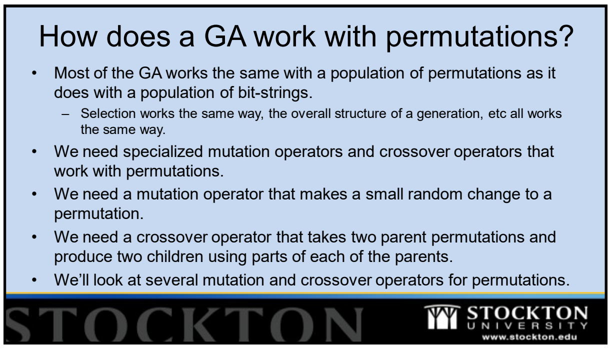
EVOLVING PERMUTATIONS

73


Permutation as the Genotype

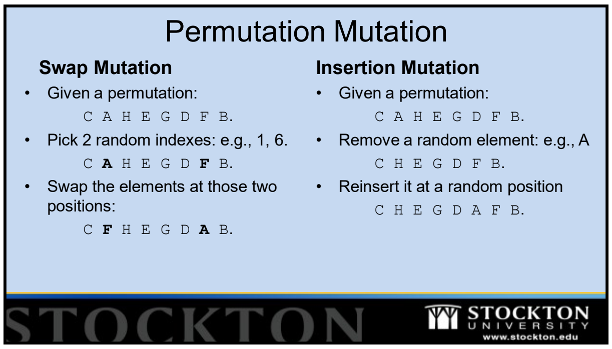
- For some optimization problems, we are searching for an optimal permutation (or ordering) of a set.
  - E.g., For the TSP, we want a permutation of the cities with lowest cost
  - Same is true for many scheduling problems, etc
- We can use bit-strings for the genotype along with all of the operators that we've already seen (see earlier presentation).
- We could also abandon bit-strings and use a more directly applicable representation as the genotype for such ordering problems.
- We can directly use a **permutation** as the genotype representation.
  - i.e., we'll have a population of permutations rather than bit-strings

74


How does a GA work with permutations?

- Most of the GA works the same with a population of permutations as it does with a population of bit-strings.
  - Selection works the same way, the overall structure of a generation, etc all works the same way.
- We need specialized mutation operators and crossover operators that work with permutations.
- We need a mutation operator that makes a small random change to a permutation.
- We need a crossover operator that takes two parent permutations and produce two children using parts of each of the parents.
- We'll look at several mutation and crossover operators for permutations.

75


Permutation Mutation

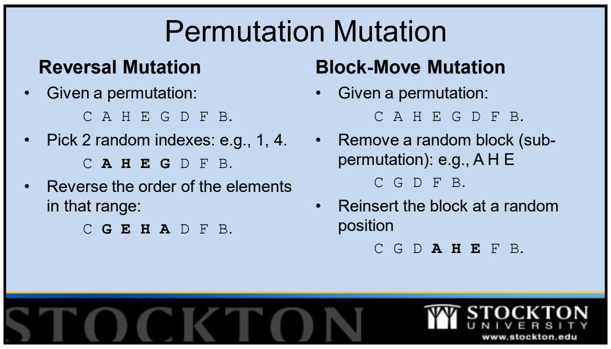
Swap Mutation

- Given a permutation:
  C A H E G D F B.
- Pick 2 random indexes: e.g., 1, 6.
  C **A** H E G D **F** B.
- Swap the elements at those two positions:
  C **F** H E G D **A** B.

Insertion Mutation

- Given a permutation:
  C A H E G D F B.
- Remove a random element: e.g., A
  C H E G D F B.
- Reinsert it at a random position
  C H E G D A F B.

76


Permutation Mutation

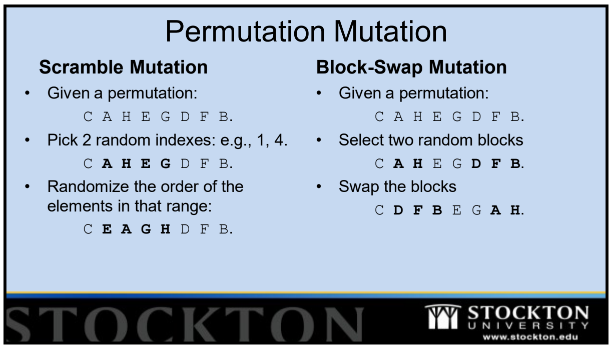
Reversal Mutation

- Given a permutation:
  C A H E G D F B.
- Pick 2 random indexes: e.g., 1, 4.
  C **A H E G** D F B.
- Reverse the order of the elements in that range:
  C **G E H A** D F B.

Block-Move Mutation

- Given a permutation:
  C A H E G D F B.
- Remove a random block (sub-permutation): e.g., A H E
  C G D F B.
- Reinsert the block at a random position
  C G D **A H E** F B.

77


Permutation Mutation

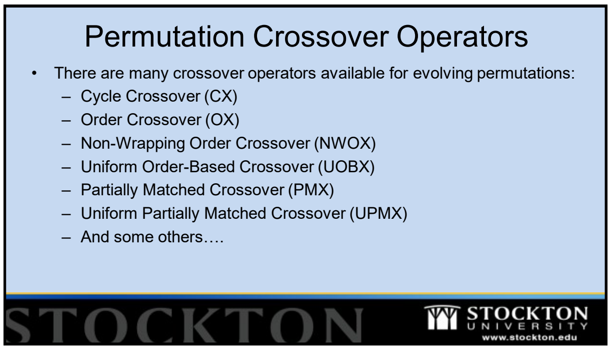
Scramble Mutation

- Given a permutation:
  C A H E G D F B.
- Pick 2 random indexes: e.g., 1, 4.
  C **A H E G** D F B.
- Randomize the order of the elements in that range:
  C **E A G H** D F B.

Block-Swap Mutation

- Given a permutation:
  C A H E G D F B.
- Select two random blocks
  C **A H** E G **D F B**.
- Swap the blocks
  C **D F B** E G **A H**.

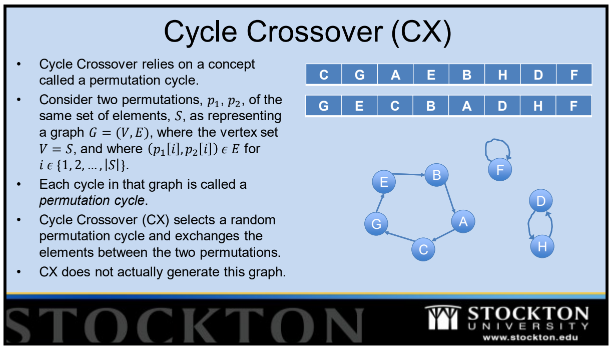
78



Permutation Crossover Operators

- There are many crossover operators available for evolving permutations:
  - Cycle Crossover (CX)
  - Order Crossover (OX)
  - Non-Wrapping Order Crossover (NWOX)
  - Uniform Order-Based Crossover (UOBX)
  - Partially Matched Crossover (PMX)
  - Uniform Partially Matched Crossover (UPMX)
  - And some others….

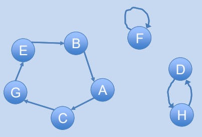
79



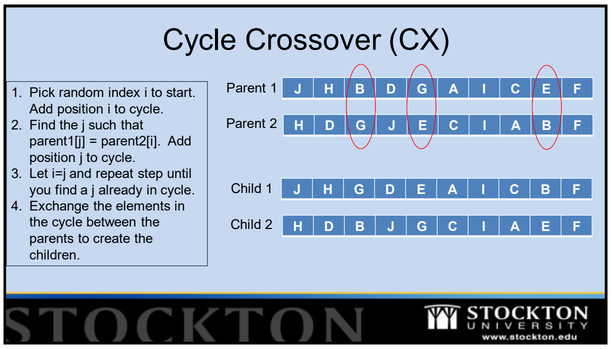
Cycle Crossover (CX)

- Cycle Crossover relies on a concept called a permutation cycle.
- Consider two permutations, p_1, p_2, of the same set of elements, S, as representing a graph $G = (V, E)$, where the vertex set $V = S$, and where $(p_1[i], p_2[i]) \in E$ for $i \in \{1, 2, \ldots, |S|\}$.
- Each cycle in that graph is called a *permutation cycle*.
- Cycle Crossover (CX) selects a random permutation cycle and exchanges the elements between the two permutations.
- CX does not actually generate this graph.

C	G	A	E	B	H	D	F
G	E	C	B	A	D	H	F



80

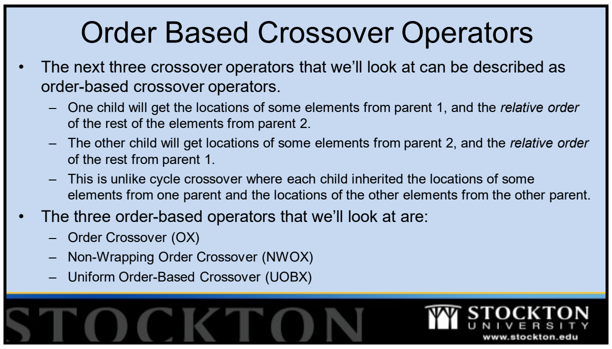


Cycle Crossover (CX)

1. Pick random index i to start. Add position i to cycle.
2. Find the j such that parent1[j] = parent2[i]. Add position j to cycle.
3. Let i=j and repeat step until you find a j already in cycle.
4. Exchange the elements in the cycle between the parents to create the children.

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F
Child 1	J	H	G	D	E	A	I	C	B	F
Child 2	H	D	B	J	G	C	I	A	E	F

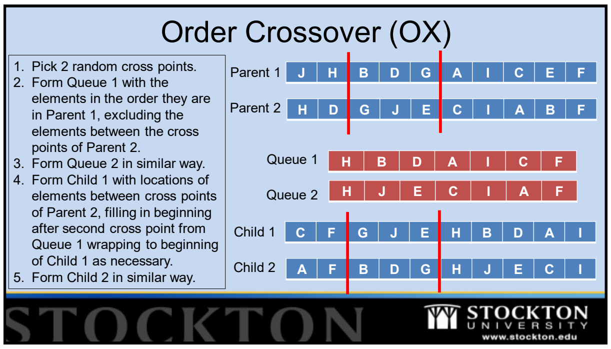
81



Order Based Crossover Operators

- The next three crossover operators that we'll look at can be described as order-based crossover operators.
  - One child will get the locations of some elements from parent 1, and the *relative order* of the rest of the elements from parent 2.
  - The other child will get locations of some elements from parent 2, and the *relative order* of the rest from parent 1.
  - This is unlike cycle crossover where each child inherited the locations of some elements from one parent and the locations of the other elements from the other parent.
- The three order-based operators that we'll look at are:
  - Order Crossover (OX)
  - Non-Wrapping Order Crossover (NWOX)
  - Uniform Order-Based Crossover (UOBX)

82



Order Crossover (OX)

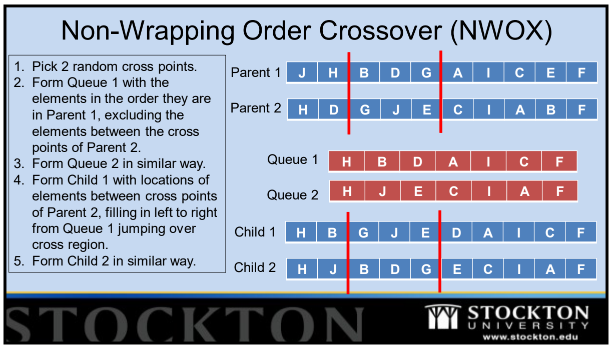
1. Pick 2 random cross points.
2. Form Queue 1 with the elements in the order they are in Parent 1, excluding the elements between the cross points of Parent 2.
3. Form Queue 2 in similar way.
4. Form Child 1 with locations of elements between cross points of Parent 2, filling in beginning after second cross point from Queue 1 wrapping to beginning of Child 1 as necessary.
5. Form Child 2 in similar way.

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F

Queue 1	H	B	D	A	I	C	F
Queue 2	H	J	E	C	I	A	F

Child 1	C	F	G	J	E	H	B	D	A	I
Child 2	A	F	B	D	G	H	J	E	C	I

83



Non-Wrapping Order Crossover (NWOX)

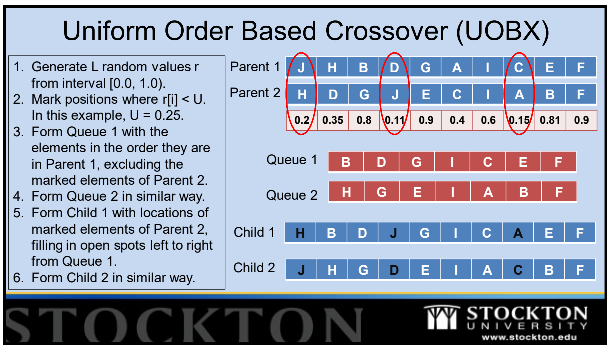
1. Pick 2 random cross points.
2. Form Queue 1 with the elements in the order they are in Parent 1, excluding the elements between the cross points of Parent 2.
3. Form Queue 2 in similar way.
4. Form Child 1 with locations of elements between cross points of Parent 2, filling in left to right from Queue 1 jumping over cross region.
5. Form Child 2 in similar way.

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F

Queue 1	H	B	D	A	I	C	F
Queue 2	H	J	E	C	I	A	F

Child 1	H	B	G	J	E	D	A	I	C	F
Child 2	H	J	B	D	G	E	C	I	A	F

84



Uniform Order Based Crossover (UOBX)

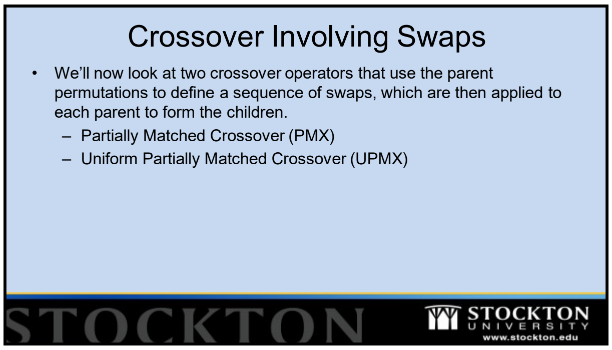
1. Generate L random values r from interval [0.0, 1.0).
2. Mark positions where r[i] < U. In this example, U = 0.25.
3. Form Queue 1 with the elements in the order they are in Parent 1, excluding the marked elements of Parent 2.
4. Form Queue 2 in similar way.
5. Form Child 1 with locations of marked elements of Parent 2, filling in open spots left to right from Queue 1.
6. Form Child 2 in similar way.

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F
	0.2	0.35	0.8	0.11	0.9	0.4	0.6	0.15	0.81	0.9

Queue 1	B	D	G	I	C	E	F
Queue 2	H	G	E	I	A	B	F

Child 1	H	B	D	J	G	I	C	A	E	F
Child 2	J	H	G	D	E	I	A	C	B	F

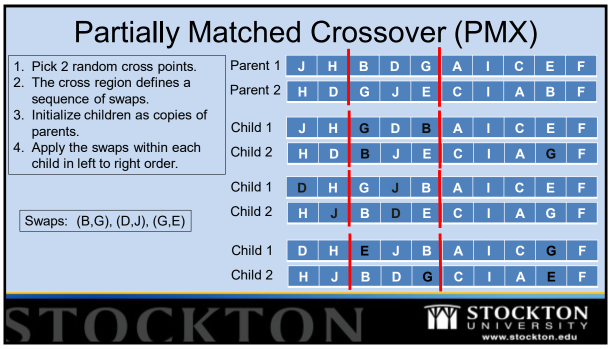
85



Crossover Involving Swaps

- We'll now look at two crossover operators that use the parent permutations to define a sequence of swaps, which are then applied to each parent to form the children.
  - Partially Matched Crossover (PMX)
  - Uniform Partially Matched Crossover (UPMX)

86



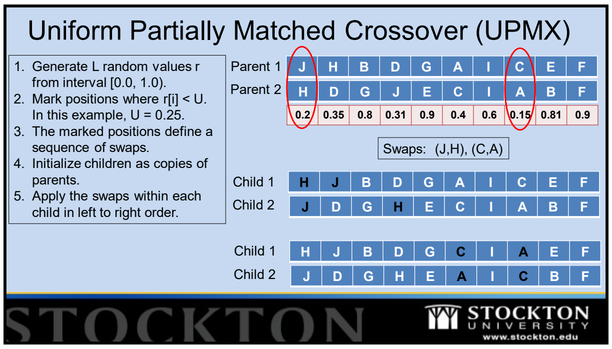
Partially Matched Crossover (PMX)

1. Pick 2 random cross points.
2. The cross region defines a sequence of swaps.
3. Initialize children as copies of parents.
4. Apply the swaps within each child in left to right order.

Swaps: (B,G), (D,J), (G,E)

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F
Child 1	J	H	G	D	B	A	I	C	E	F
Child 2	H	D	B	J	E	C	I	A	G	F
Child 1	D	H	G	J	B	A	I	C	E	F
Child 2	H	J	B	D	E	C	I	A	G	F
Child 1	D	H	E	J	B	A	I	C	G	F
Child 2	H	J	B	D	G	C	I	A	E	F

87



Uniform Partially Matched Crossover (UPMX)

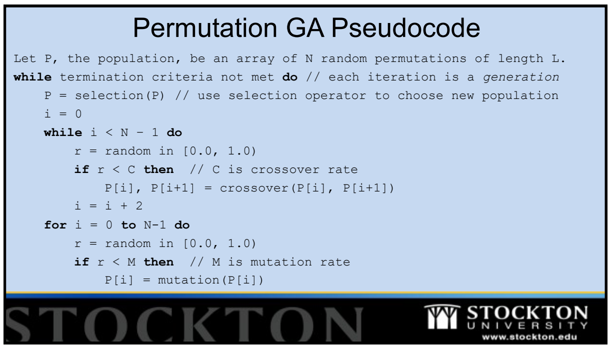
1. Generate L random values r from interval [0.0, 1.0).
2. Mark positions where r[i] < U. In this example, U = 0.25.
3. The marked positions define a sequence of swaps.
4. Initialize children as copies of parents.
5. Apply the swaps within each child in left to right order.

Parent 1	J	H	B	D	G	A	I	C	E	F
Parent 2	H	D	G	J	E	C	I	A	B	F
	0.2	0.35	0.8	0.31	0.9	0.4	0.6	0.15	0.81	0.9

Swaps: (J,H), (C,A)

Child 1	H	J	B	D	G	A	I	C	E	F
Child 2	J	D	G	H	E	C	I	A	B	F
Child 1	H	J	B	D	G	C	I	A	E	F
Child 2	J	D	G	H	E	A	I	C	B	F

88



Permutation GA Pseudocode

```
Let P, the population, be an array of N random permutations of length L.
while termination criteria not met do // each iteration is a generation
    P = selection(P) // use selection operator to choose new population
    i = 0
    while i < N - 1 do
        r = random in [0.0, 1.0)
        if r < C then  // C is crossover rate
            P[i], P[i+1] = crossover(P[i], P[i+1])
        i = i + 2
    for i = 0 to N-1 do
        r = random in [0.0, 1.0)
        if r < M then  // M is mutation rate
            P[i] = mutation(P[i])
```

89

Lesson 10

EVOLUTION STRATEGIES

90

Evolution Strategies

- Evolution Strategy is another of the various Evolutionary Computation algorithms.
- Evolution Strategy is the name of the class of Evolutionary Computation algorithms focused on real-valued optimization (rather than discrete optimization problems).
- Example 1: Finding the minimum or maximum of some multivariate, non-linear function, $f(x_1, x_2, \dots, x_N)$.
- Example 2: Optimizing the control parameters (e.g., C, M, etc) of a GA.

91

Can't We Just Use a GA for This?

- Yes, you can use a GA.
- Can easily encode real-values (approximated with floating-point numbers) in binary—thus, using bit-strings.
  - E.g., IEEE's floating-point standard
- Issue 1: Crossover at cross points in the middle of a sub-string of bits representing a floating-point number
  - Wouldn't really make sense.
  - The minimum defining length of a schema that makes sense in this application is 32bits (for single-precision floating point numbers or 64bits (for double-precision)
- Issue 2: Mutation tends to make big jumps (even 1-bit mutations can drastically change magnitude of floating-point number)

92

Evolution Strategy

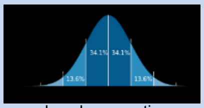
- An Evolution Strategy uses a vector of Reals as the genotype rather than a bit-string
  - Or whatever the closest equivalent is in the implementation language (usually floating-point numbers)
  - Either a vector (or array) of doubles in Java
- The population is then a population of arrays of doubles.
- Crossover:
  - The obvious extension of the bit-string crossover operators, just operating on arrays of floating-point numbers instead of arrays of bits
  - Can use single point, two-point, k-point, uniform, etc
- What about mutation?

93

Gaussian Mutation

- Gaussian Mutation is the most commonly used mutation operator for ES.
- Let V, a vector of reals, be a member of the population.
- We can mutate each element V[i] of V as follows:
  - Assume we have lower and upper limits on range for V[i].
  - Let g be a random value selected from a Gaussian distribution (i.e., Normal distribution) with mean 0, and standard deviation σ.
  - V[i] = V[i] + g
  - If V[i] < lowerLimit then V[i] = lowerLimit
  - If V[i] > upperLimit then V[i] = upperLimit
- Standard deviation usually set high at start, and decayed each generation (like the temperature simulated annealing)

```
Python: g = random.gauss(0, σ)
Java: Random r = …;
        g = σ * r.nextGaussian();
```



94

Lesson 11

GENETIC PROGRAMMING

95

Genetic Programming

- **Genetic Programming (GP)** is a form of evolutionary computation applied to Automatic Inductive Programming
- What is Automatic Inductive Programming?
  - **Automatic**: Working by itself with little or no direct control
  - **Inductive**: Inference of general laws from specific examples
  - **Programming**: Well, you know what that is.
- Automatic inductive programming needs examples of inputs with corresponding outputs (e.g., a set of test cases --- think unit testing)
- Derives a computer program that passes that set of test cases
- Need a good, comprehensive set of test cases.

96

Computational Modeling: Example Application

- Modeling some natural or artificial process
  - Collect some sensor (or other) data.
  - Decide what it is you are trying to predict.
  - The data measuring what you want to predict is your output.
  - The other sensor data are your inputs.
  - You want to derive a program that can make your predictions.
- Example: Predicting some environmental phenomena
  - Inputs: Sensor readings (temperature measurements, barometric pressure, etc)
  - Output: Whatever phenomena you are trying to predict (e.g., rain)

97

Automated System Control: Example Application

- Automated control of some complex system
  - Gather some input sensor data (systems percepts of its environment).
  - Determine corresponding desired output data (commands and other inputs to actuators)
  - You want to derive a program that maps input sensor data to the correct actuator settings.
- Example: Automated Control of something normally human controlled
  - Inputs: Sensor readings (radar, laser, pixels from camera images, etc)
  - Outputs: Configuration for actuators (e.g., position of controls, etc)
  - Imagine trying to learn a navigation system for an automated car or other vehicle

98

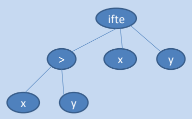
Genetic Programming (GP)

- Genetic Programming (GP) is one form of automatic inductive programming
- Created by John Koza around 1990.
  - Koza has actually used the approach to successfully evolve patentable inventions (e.g., to evolve circuit board designs)
  - Koza used a massively parallel system in much of his GP work (e.g., a Beowulf-style cluster with thousands of nodes).
- GP extends the concepts of GA to evolving computer programs
  - No bit-strings
  - Evolves program trees using specialized operators
  - Much larger search space compared to the GA
  - Usually requires much larger population sizes, etc

99

Genotype: Expression Trees

- GP uses expression trees as the genotype (and not bit-strings)
- You can easily generate a prefix notation expression from one of these trees (e.g., mapping genotype to phenotype)
  - GP originally formulated for Lisp, which uses prefix notation



(ifte (> x y) x y)

100

GP Expression Trees

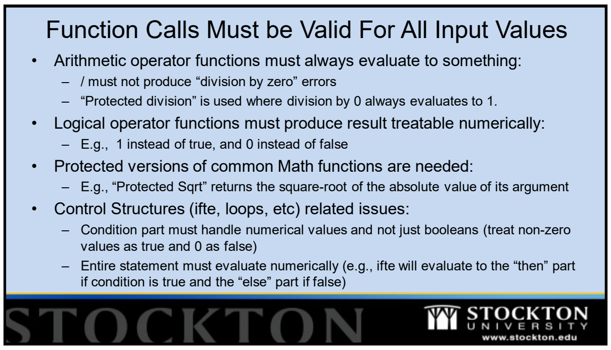
- Terminal nodes of the tree (i.e., leaves of the tree):
  - Variables: X, Y, Z, etc
  - Constants: 1, 2, 3, 2.3, 4.56, etc
  - Zero-Argument Functions (i.e., function with no inputs):
    - E.g., rand() might be a function that returns a random floating-point value between 0 and 1
    - E.g., perhaps a function that when called polls a sensor to get the current reading

101

GP Expression Trees

- Interior nodes are function calls with 1 or more inputs (i.e., parameters)
- A node for a function with k inputs will have k children in the tree
- GP originated with the Lisp language (a "functional" language), thus everything other than data treated as functions
- Function Examples:
  - Arithmetic operators: +, -, *, /
  - Logical operators: >, <, =, etc
  - Common Math functions: cos, sin, sqrt, etc
  - Programming control structures: if, ifte (" if..then..else"), loops, etc:
    - ifte has 3 parameters (condition, value if condition true, value if condition false)
  - Calls to evolved functions (more advanced GP technique derives sub-procedures)

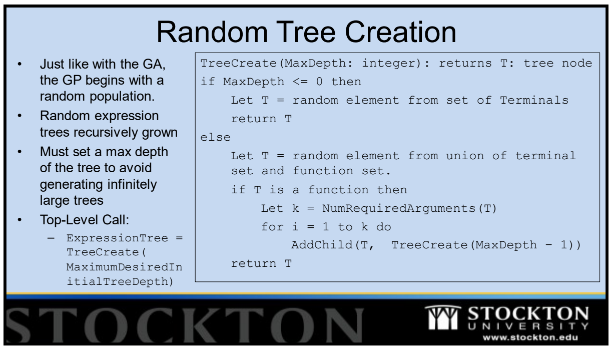
102



Function Calls Must be Valid For All Input Values

- Arithmetic operator functions must always evaluate to something:
  - / must not produce "division by zero" errors
  - "Protected division" is used where division by 0 always evaluates to 1.
- Logical operator functions must produce result treatable numerically:
  - E.g., 1 instead of true, and 0 instead of false
- Protected versions of common Math functions are needed:
  - E.g., "Protected Sqrt" returns the square-root of the absolute value of its argument
- Control Structures (ifte, loops, etc) related issues:
  - Condition part must handle numerical values and not just booleans (treat non-zero values as true and 0 as false)
  - Entire statement must evaluate numerically (e.g., ifte will evaluate to the "then" part if condition is true and the "else" part if false)

103

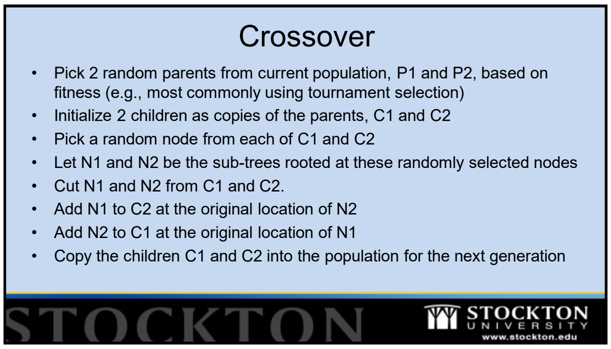


Random Tree Creation

- Just like with the GA, the GP begins with a random population.
- Random expression trees recursively grown
- Must set a max depth of the tree to avoid generating infinitely large trees
- Top-Level Call:
  - ExpressionTree = TreeCreate(MaximumDesiredInitialTreeDepth)

```
TreeCreate(MaxDepth: integer): returns T: tree node
if MaxDepth <= 0 then
    Let T = random element from set of Terminals
    return T
else
    Let T = random element from union of terminal
    set and function set.
    if T is a function then
        Let k = NumRequiredArguments(T)
        for i = 1 to k do
            AddChild(T,  TreeCreate(MaxDepth – 1))
    return T
```

104

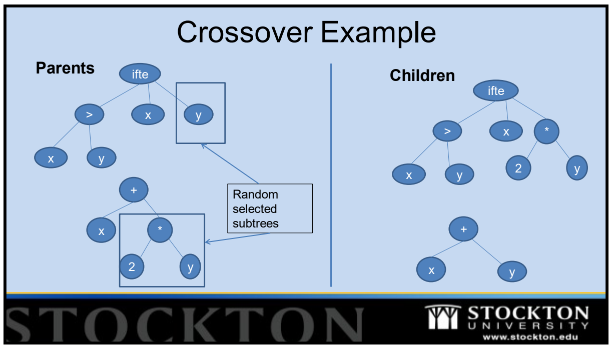


Crossover

- Pick 2 random parents from current population, P1 and P2, based on fitness (e.g., most commonly using tournament selection)
- Initialize 2 children as copies of the parents, C1 and C2
- Pick a random node from each of C1 and C2
- Let N1 and N2 be the sub-trees rooted at these randomly selected nodes
- Cut N1 and N2 from C1 and C2.
- Add N1 to C2 at the original location of N2
- Add N2 to C1 at the original location of N1
- Copy the children C1 and C2 into the population for the next generation

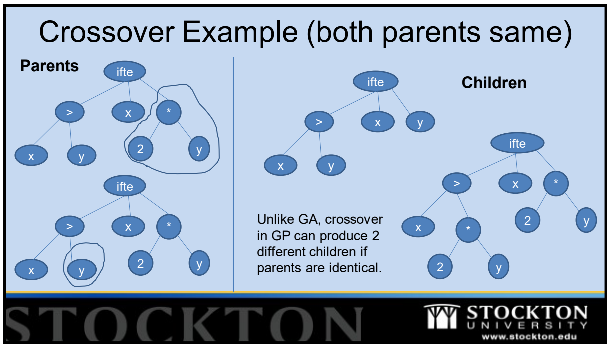
105

Crossover Example



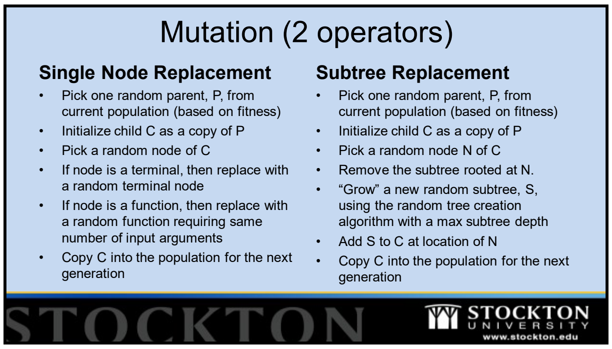
106

Crossover Example (both parents same)



Unlike GA, crossover in GP can produce 2 different children if parents are identical.

107



Mutation (2 operators)

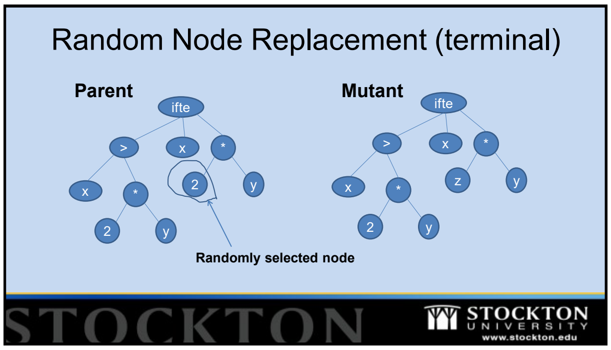
Single Node Replacement

- Pick one random parent, P, from current population (based on fitness)
- Initialize child C as a copy of P
- Pick a random node of C
- If node is a terminal, then replace with a random terminal node
- If node is a function, then replace with a random function requiring same number of input arguments
- Copy C into the population for the next generation

Subtree Replacement

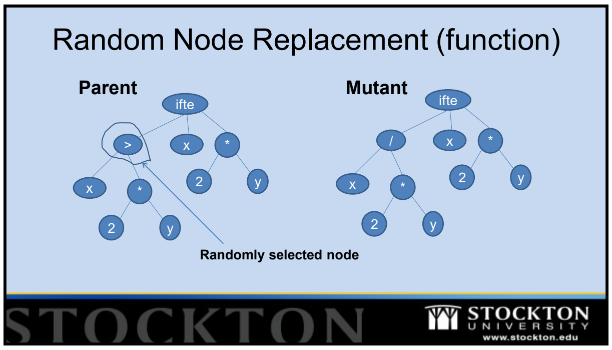
- Pick one random parent, P, from current population (based on fitness)
- Initialize child C as a copy of P
- Pick a random node N of C
- Remove the subtree rooted at N.
- "Grow" a new random subtree, S, using the random tree creation algorithm with a max subtree depth
- Add S to C at location of N
- Copy C into the population for the next generation

108



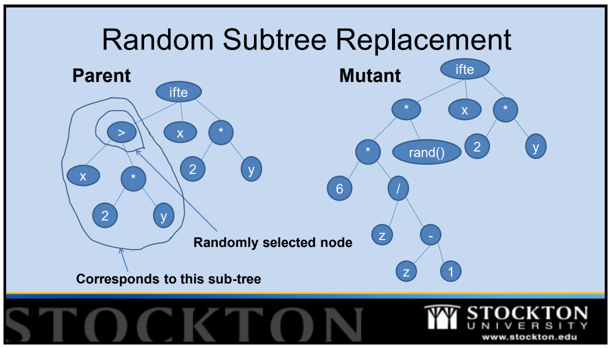
Random Node Replacement (terminal)

109



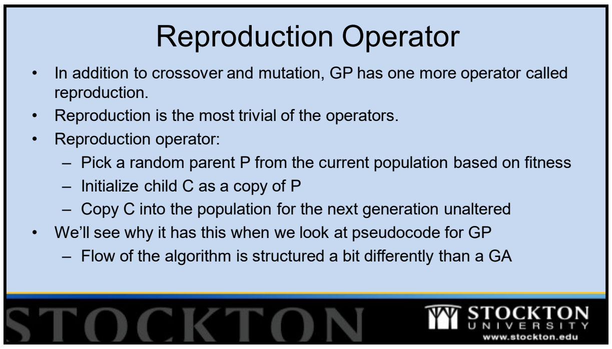
Random Node Replacement (function)

110



Random Subtree Replacement

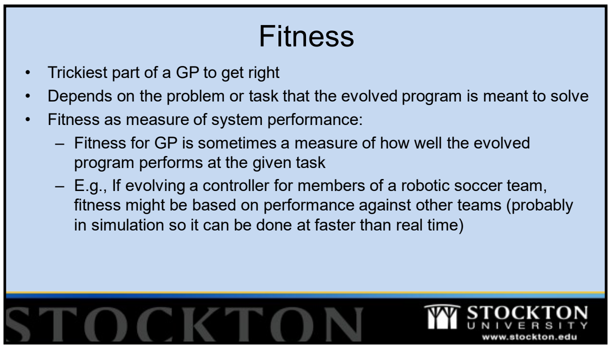
111



Reproduction Operator

- In addition to crossover and mutation, GP has one more operator called reproduction.
- Reproduction is the most trivial of the operators.
- Reproduction operator:
  - Pick a random parent P from the current population based on fitness
  - Initialize child C as a copy of P
  - Copy C into the population for the next generation unaltered
- We'll see why it has this when we look at pseudocode for GP
  - Flow of the algorithm is structured a bit differently than a GA

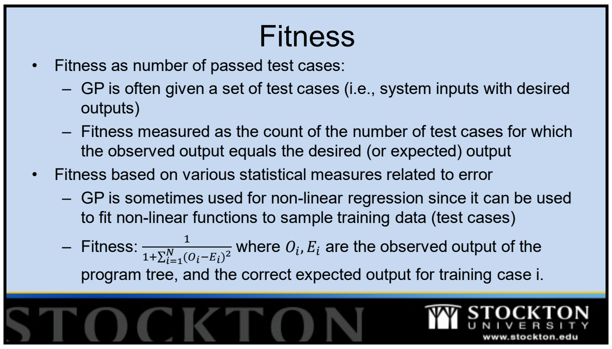
112



Fitness

- Trickiest part of a GP to get right
- Depends on the problem or task that the evolved program is meant to solve
- Fitness as measure of system performance:
  - Fitness for GP is sometimes a measure of how well the evolved program performs at the given task
  - E.g., If evolving a controller for members of a robotic soccer team, fitness might be based on performance against other teams (probably in simulation so it can be done at faster than real time)

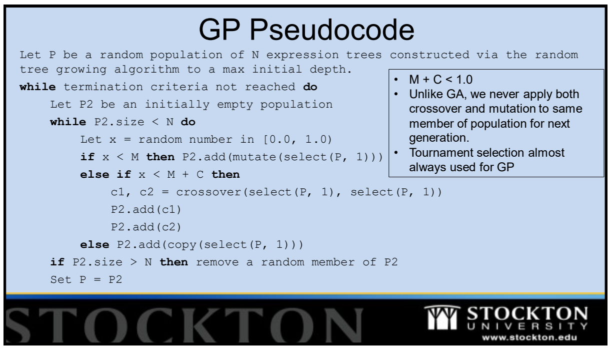
113



Fitness

- Fitness as number of passed test cases:
  - GP is often given a set of test cases (i.e., system inputs with desired outputs)
  - Fitness measured as the count of the number of test cases for which the observed output equals the desired (or expected) output
- Fitness based on various statistical measures related to error
  - GP is sometimes used for non-linear regression since it can be used to fit non-linear functions to sample training data (test cases)
  - Fitness: $\frac{1}{1+\sum_{i=1}^{N}(O_i-E_i)^2}$ where O_i, E_i are the observed output of the program tree, and the correct expected output for training case i.

114



GP Pseudocode

```
Let P be a random population of N expression trees constructed via the random
tree growing algorithm to a max initial depth.
while termination criteria not reached do
    Let P2 be an initially empty population
    while P2.size < N do
        Let x = random number in [0.0, 1.0)
        if x < M then P2.add(mutate(select(P, 1)))
        else if x < M + C then
            c1, c2 = crossover(select(P, 1), select(P, 1))
            P2.add(c1)
            P2.add(c2)
        else P2.add(copy(select(P, 1)))
    if P2.size > N then remove a random member of P2
    Set P = P2
```

- M + C < 1.0
- Unlike GA, we never apply both crossover and mutation to same member of population for next generation.
- Tournament selection almost always used for GP

115