# 5: Genetic Algorithm

- GA history and overview
- Holland's simple GA
- A SGA example
- SGA discussion
- SGA improvement
- Textbook Chapter 4.1, 4.2, 6.1

Prof. Ting Hu, School of Computing, Queen's University

# What are the different types of EAs

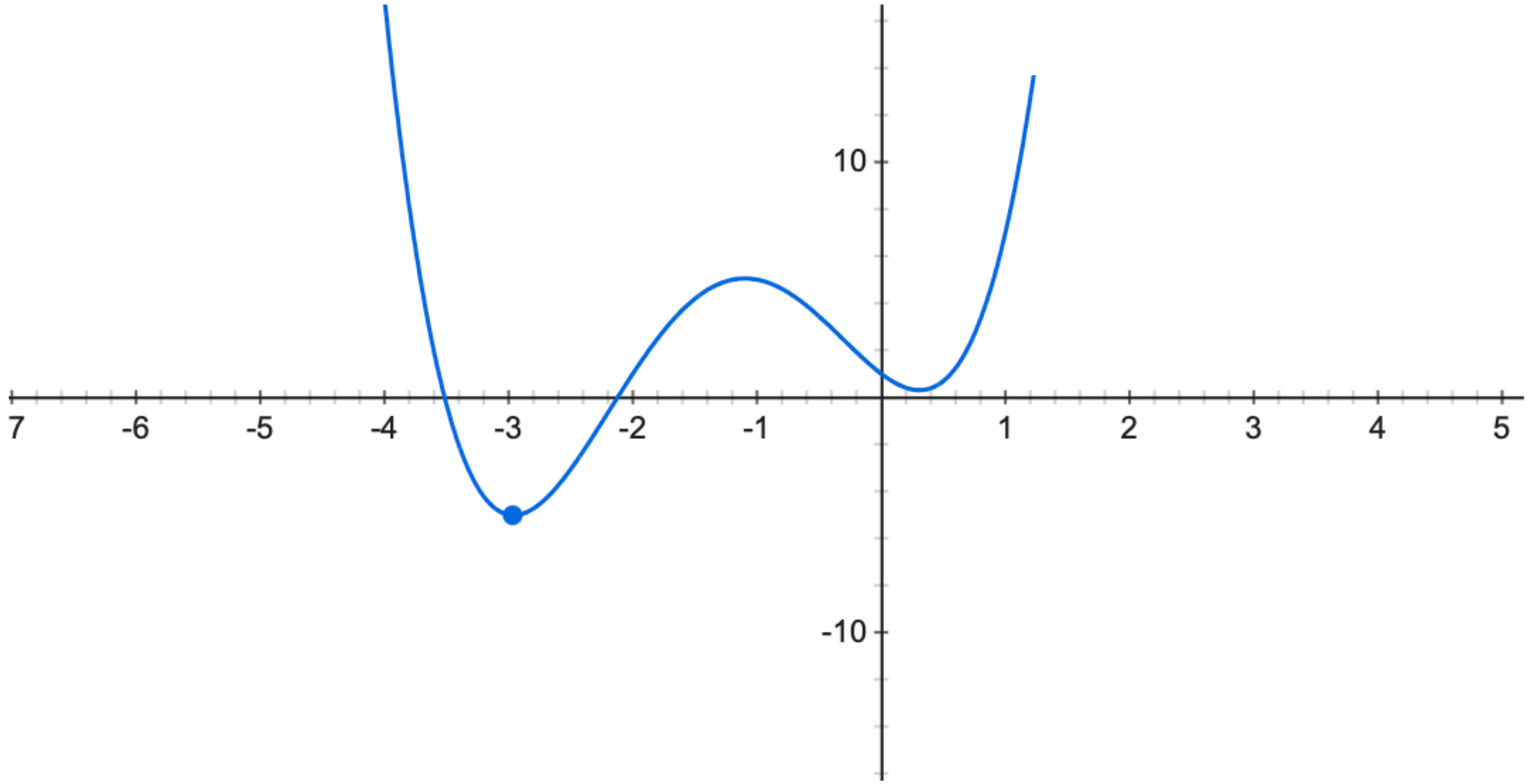
- Historically different flavors of EAs have been associated with different representations
  - Binary strings: Genetic Algorithms
  - Real-valued vectors: Evolution Strategies
  - Finite state machines: Evolutionary Programming
  - LISP trees: Genetic Programming
- These differences are largely irrelevant, best strategies
  - choose representation to suit problem
  - choose variation operators to suit representation
- Selection operators only use fitness and so are <u>independent</u> of representation

# GA quick overview

- Developed: USA in the 1970's
  - *Adaptation in Natural and Artificial Systems*, John Holland, 1975
- Early names: J. Holland, E. Goodman, K. DeJong, D. Goldberg
- Typically applied to:
  - discrete function optimization problems
  - benchmark problems
  - straightforward problems using a binary representation
- Features:
  - not very fast
  - good heuristic for combinatorial problems
  - traditionally emphasizes combining information from good parents
  - many variants, e.g., reproduction models, operators

# An example function optimization problem

$$\min f(x), f(x) = x^4 + 5x^3 + 4x^2 - 4x + 1$$



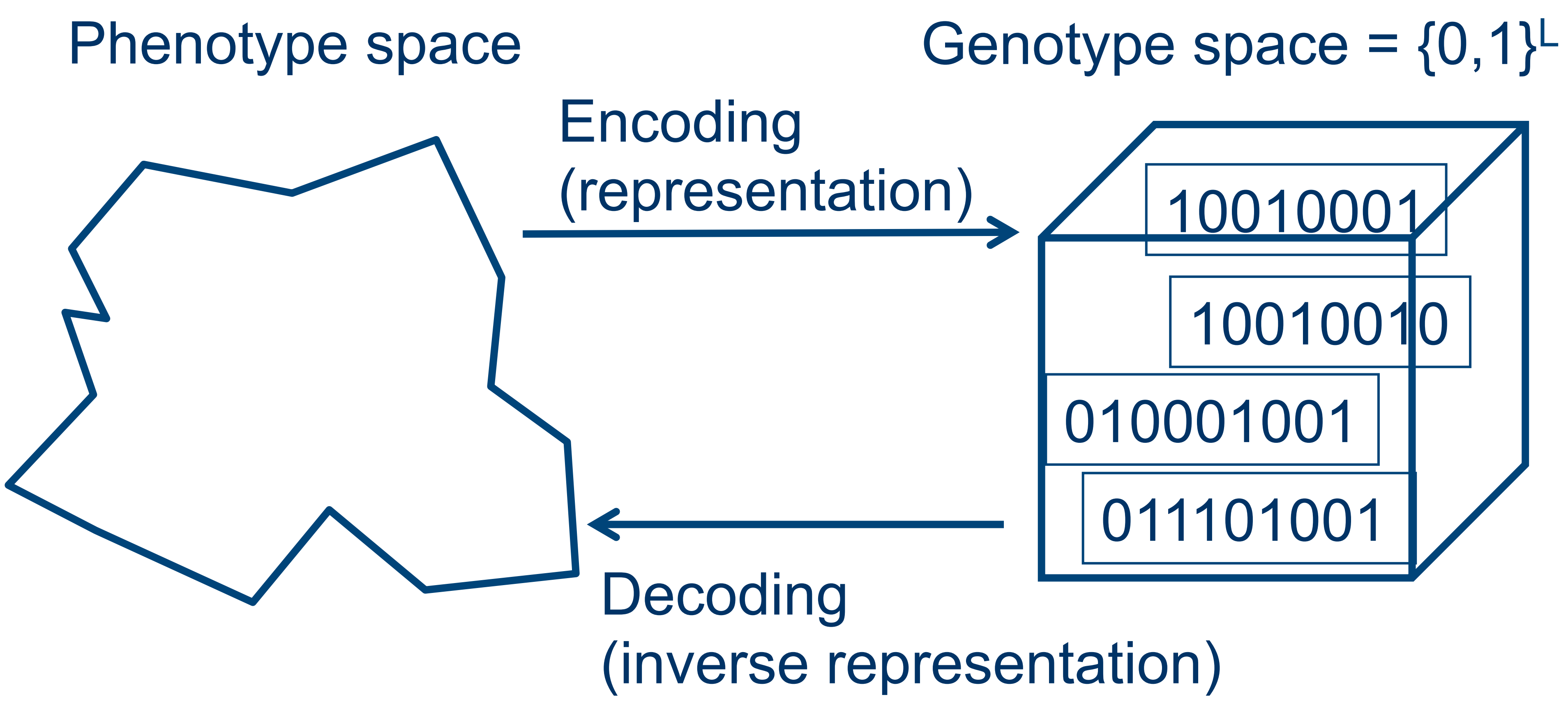
# Genetic Algorithms

- Holland’s original GA is now known as the simple genetic algorithm (SGA)
- Other GAs use different:
  - representations
  - mutations
  - crossovers
  - selection mechanisms
  - population management

# SGA technical summary

| | |
|---|---|
| **Representation** | Binary strings |
| **Mutation** | Bit flip |
| **Recombination** | One-point crossover |
| **Parent selection** | Fitness proportional - implemented using Roulette Wheel |
| **Survivor selection** | Generational |

# SGA representation

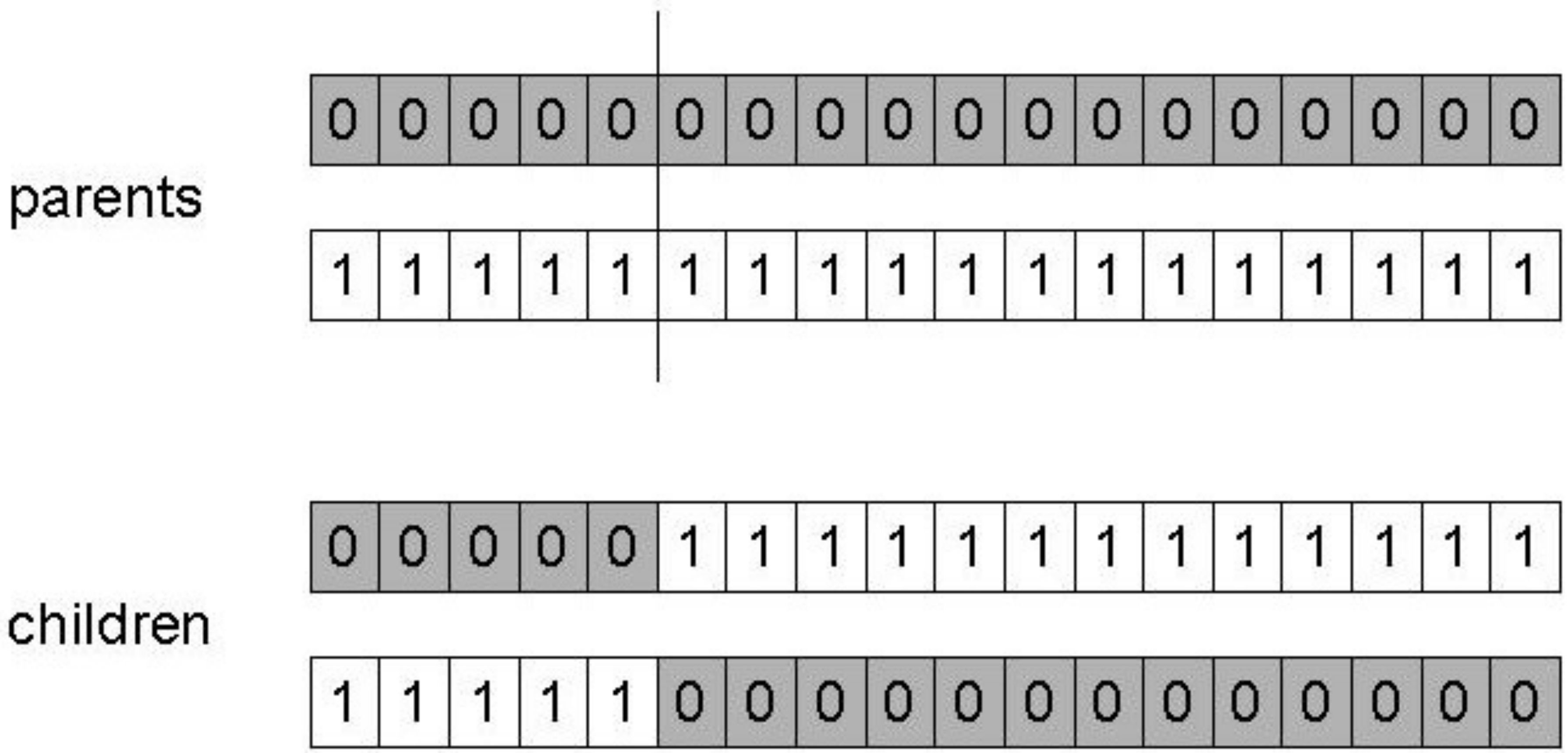


# SGA iteration

1. Select parents for the mating pool (size of mating pool = population size)
2. Shuffle the mating pool
3. Apply crossover tor each consecutive pair with probability $p_c$, otherwise copy parents
4. Apply mutation to each offspring (bit-flip with probability $p_m$ independently for each bit)
5. Replace the whole population with the resulting offspring

# SGA operators: one-point crossover

- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails
- $p_c$ typically in range (0.6, 0.9)



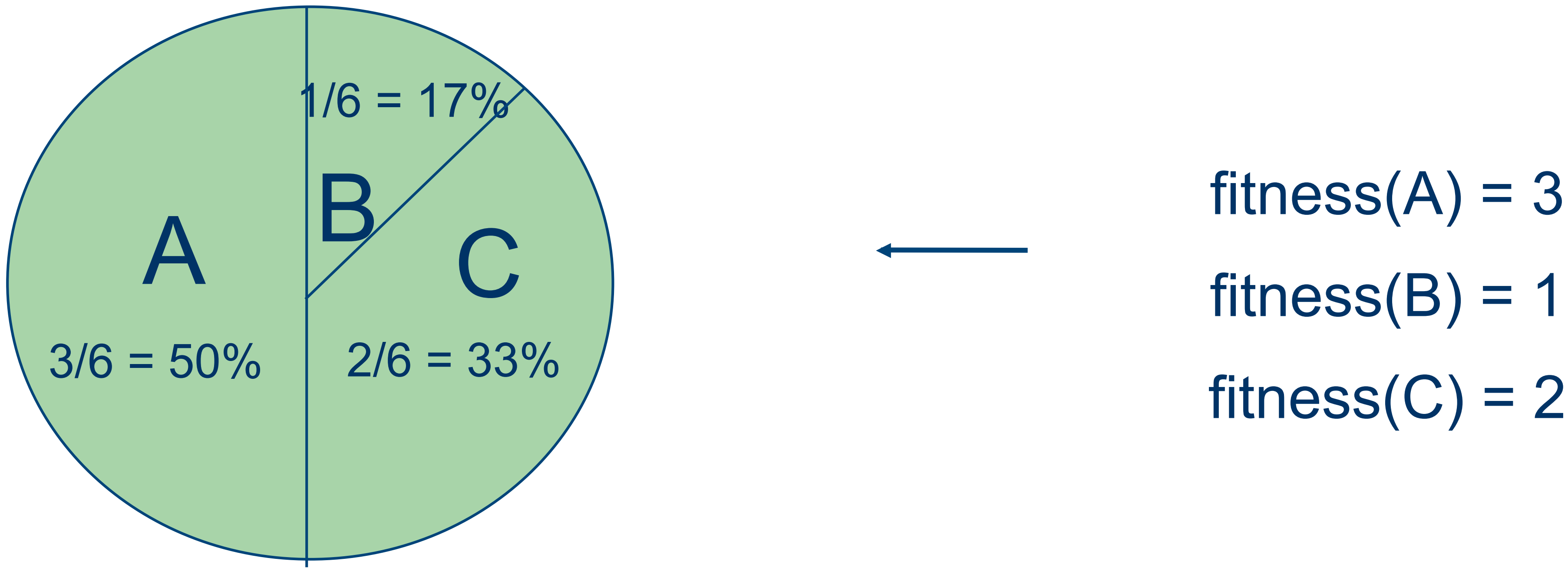
# SGA operators: mutation

- Alter each gene independently with a probability $p_m$
- $p_m$ is called the mutation rate
  - depends on the desired outcome (*all* good members or *one* highly fit individual?)
  - typically one gene per offspring

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| parent | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| child | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

# SGA operators: parent selection

- Main idea: better individuals get higher chance to reproduce
  - changes proportional to fitness
  - implementation: roulette wheel technique
    - Assign to each individual a part of the roulette wheel
    - Spin wheel *n* times to select *n* individuals



# Example

- Maximize $x^2$ over $\{0, 1, 2, \ldots, 31\}$
- GA approach
  - representation: binary code, 5 bits
  - population size: 4
  - one-point crossover, bitwise mutation
  - roulette wheel selection
  - random initialization

# Example: parent selection

| String no. | Initial population | $x$ Value | Fitness $f(x) = x^2$ | $Prob_i$ | Expected count | Actual count |
|---|---|---|---|---|---|---|
| 1 | 0 1 1 0 1 | 13 | 169 | 0.14 | 0.58 | 1 |
| 2 | 1 1 0 0 0 | 24 | 576 | 0.49 | 1.97 | 2 |
| 3 | 0 1 0 0 0 | 8 | 64 | 0.06 | 0.22 | 0 |
| 4 | 1 0 0 1 1 | 19 | 361 | 0.31 | 1.23 | 1 |
| Sum | | | 1170 | 1.00 | 4.00 | 4 |
| Average | | | 293 | 0.25 | 1.00 | 1 |
| Max | | | 576 | 0.49 | 1.97 | 2 |

# Example: crossover

| String no. | Mating pool | Crossover point | Offspring after xover | $x$ Value | Fitness $f(x) = x^2$ |
|---|---|---|---|---|---|
| 1 | 0 1 1 0 \| 1 | 4 | 0 1 1 0 0 | 12 | 144 |
| 2 | 1 1 0 0 \| 0 | 4 | 1 1 0 0 1 | 25 | 625 |
| 2 | 1 1 \| 0 0 0 | 2 | 1 1 0 1 1 | 27 | 729 |
| 4 | 1 0 \| 0 1 1 | 2 | 1 0 0 0 0 | 16 | 256 |
| Sum | | | | | 1754 |
| Average | | | | | 439 |
| Max | | | | | 729 |

# Example: mutation

| String no. | Offspring after xover | Offspring after mutation | $x$ Value | Fitness $f(x) = x^2$ |
|---|---|---|---|---|
| 1 | 0 1 1 0 0 | $\boxed{1}$ 1 1 0 0 | 26 | 676 |
| 2 | 1 1 0 0 1 | 1 1 0 0 1 | 25 | 625 |
| 2 | 1 1 0 1 1 | 1 1 0 1 1 | 27 | 729 |
| 4 | 1 0 0 0 0 | 1 0 $\boxed{1}$ 0 0 | 18 | 324 |
| Sum | | | | 2354 |
| Average | | | | 588.5 |
| Max | | | | 729 |

# Simple GA

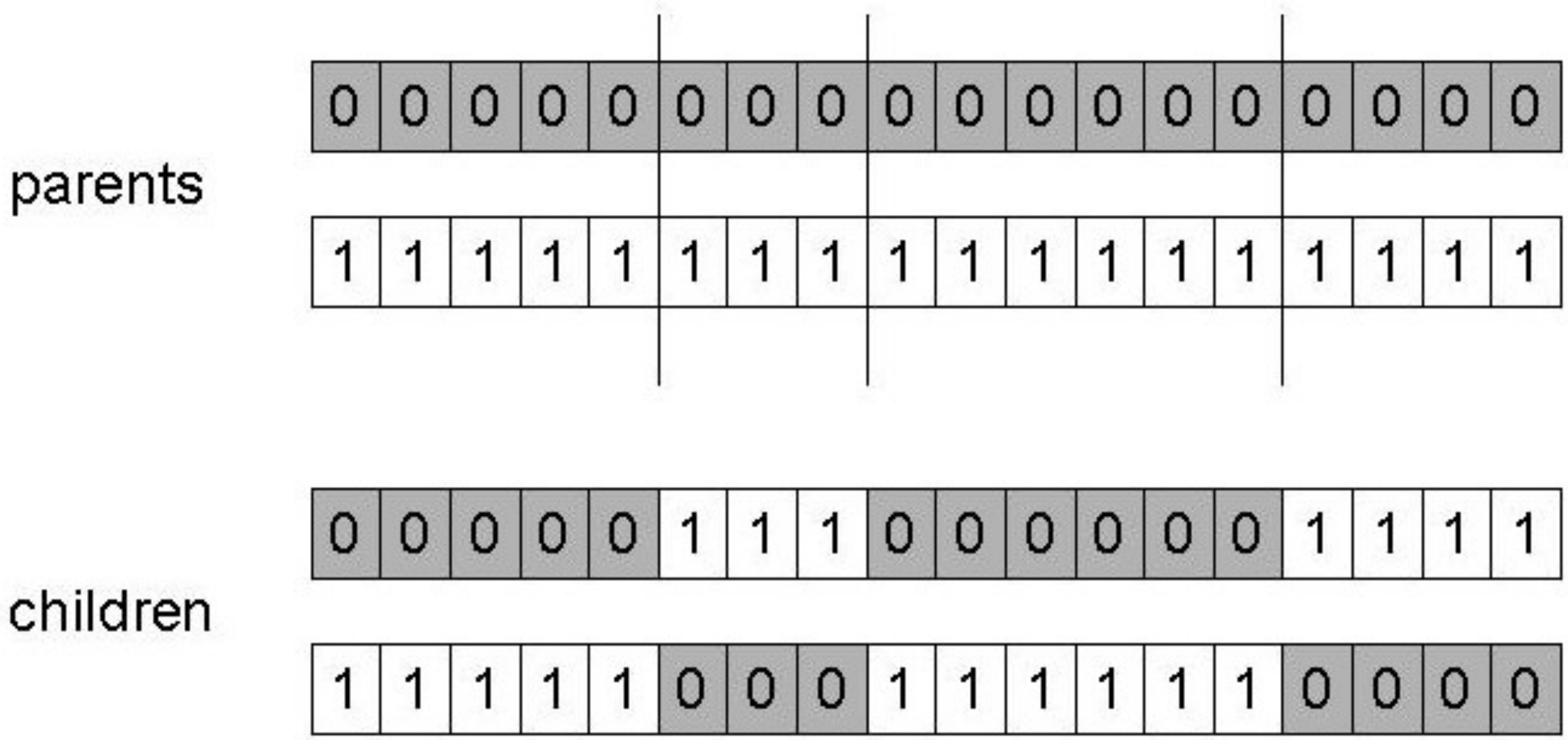
- Has been subject of many (early) studies
  - still often used as benchmark for novel GAs
- Show many shortcomings
  - representation is too restrictive
  - mutation & crossover only applicable for bit-string integer representations
  - selection mechanism sensitive for converging populations with close fitness values
  - generational population model can be improved with explicit survivor selection

# Alternative crossover operators

- Performance with one-point crossover depends on the order that variables occur in the representation
    - more likely to keep together genes that are near each other
    - can never keep together genes from opposite ends of string
    - known as *positional bias*
    - can be exploited if we know about the structure of our problem, but this is not usually the case

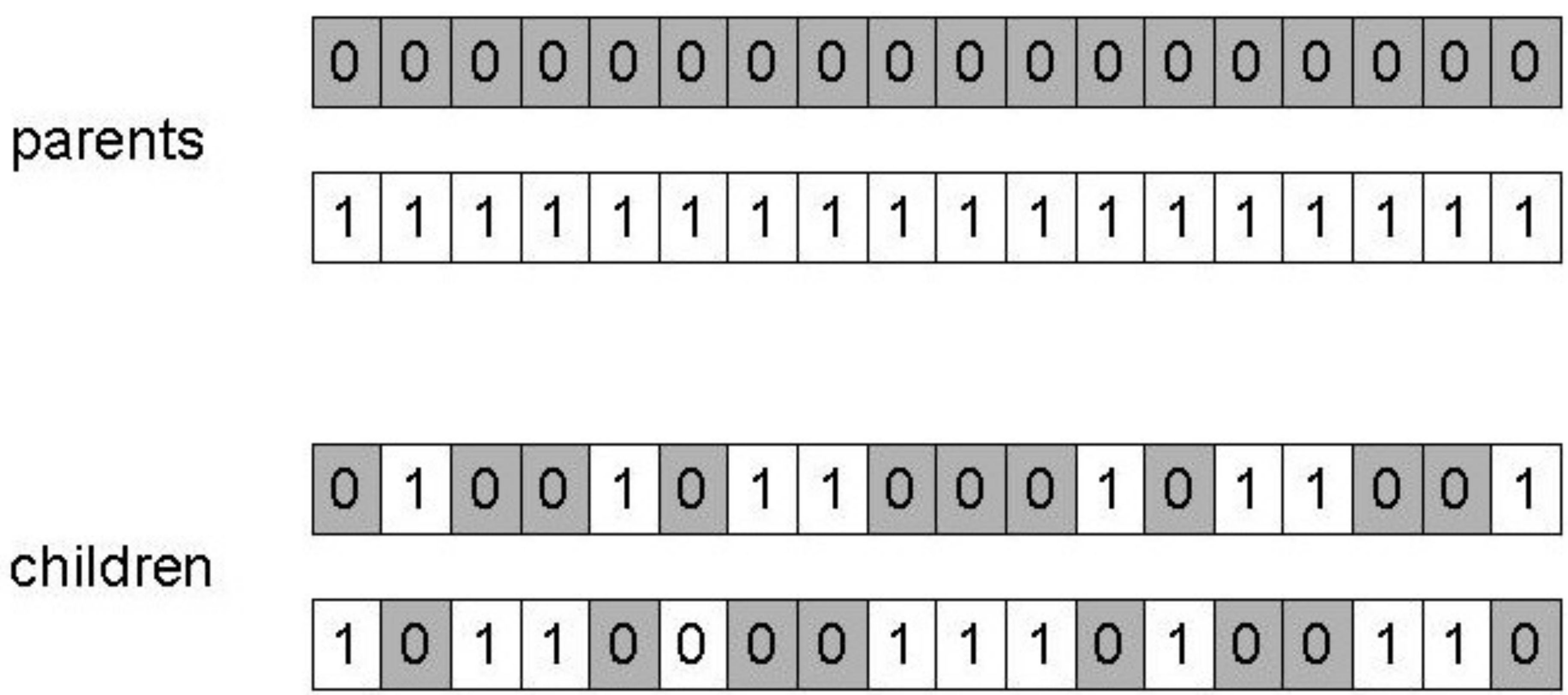
## n-point crossover

- Choose *n* random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalization of one-point crossover



# Uniform crossover

- Assign "heads" to one parent, "tails" to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position



# Which crossover operator to use?

- Impossible to state that one or the other of those operators performs best on any given problem
- Important to understand the types of bias exhibited by different recombination operators
- Know patterns or dependencies in the chosen representation when designing an algorithm for a particular problem

# Crossover or mutation?

- Decade long debate: which one is better / necessary / main background search operator

- Answer (at least, wide agreement):
    - it depends on the problem, but
    - in general, it is good to have both
    - mutation-only-EA is possible, xover-only-EA would not work

# Crossover or mutation?

- **Exploration**: discovering promising areas in the search space, i.e., gaining information on the problem
- **Exploitation**: optimizing within a promising area, i.e., using information
- There is co-operation *and* competition between them
    - crossover is explorative, it makes a big jump to an area somewhere “in between” two (parent) areas
    - mutation is exploitative, it creates random small diversions, thereby staying near (in the area of) the parent

# Crossover or mutation?

- Only crossover can combine information from two parents
- Only mutation can introduce new information (alleles)
- Crossover does not change the allele frequencies of the population

# The fundamental tension

- Diversity vs. convergence
- Variation operators, especially mutations, increase diversity, ensuring better search and preventing getting stuck in local optima
- GA can possibly make progress as  long as it is not converged
- However, we cannot wait "forever" for a GA to converge, as we are seeking solutions
- Balancing these issues is the key

# When to use a GA?

- Highly multimodal functions
- Discrete or discontinuous functions
- High-dimensionality functions, including many combinatorial ones
- Nonlinear dependencies on parameters (interactions)
- When not to use a GA?