Local Search
[RN2] Section 4.3
[RN3] Section 4.1

CS 486/686

University of Waterloo

Lecture 5: May 15, 2017

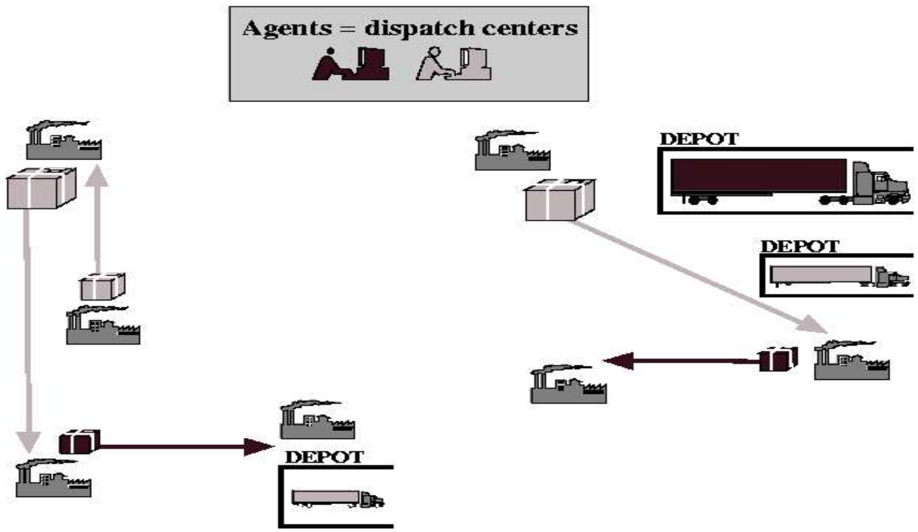
Outline

- Iterative improvement algorithms
- Hill climbing search
- Simulated annealing
- Genetic algorithms

Introduction

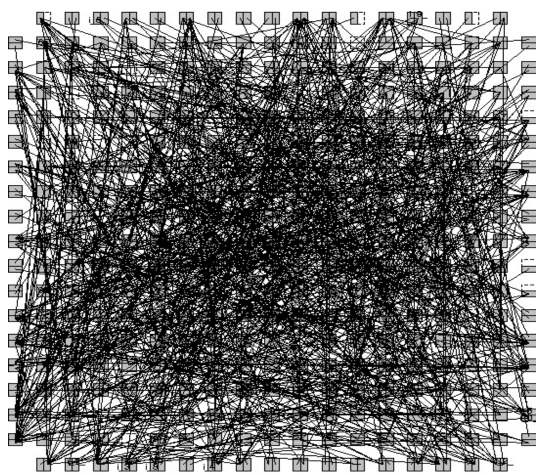
- So far we have studied algorithms which systematically explore search spaces
  - Keep one or more paths in memory
  - When the goal is found, the solution consists of a path to the goal
- For many problems the path is unimportant

Examples

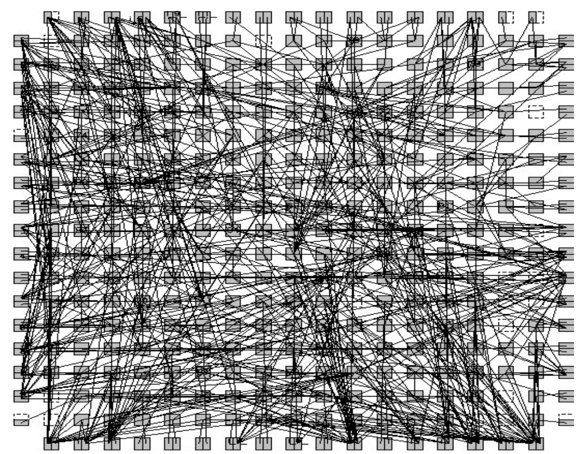


Vehicle routing

Channel Routing

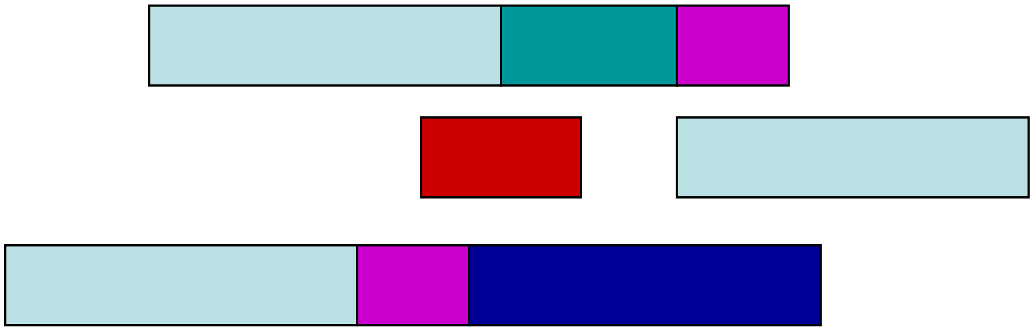


Initial Placement. Cost: 74.5562. Channel Factor: 100



Final Placement. Cost: 28.5384. Channel Factor: 100

Examples



Job shop scheduling

A v ~B v C

~A v C v D

B v D v ~E

~C v ~D v ~E

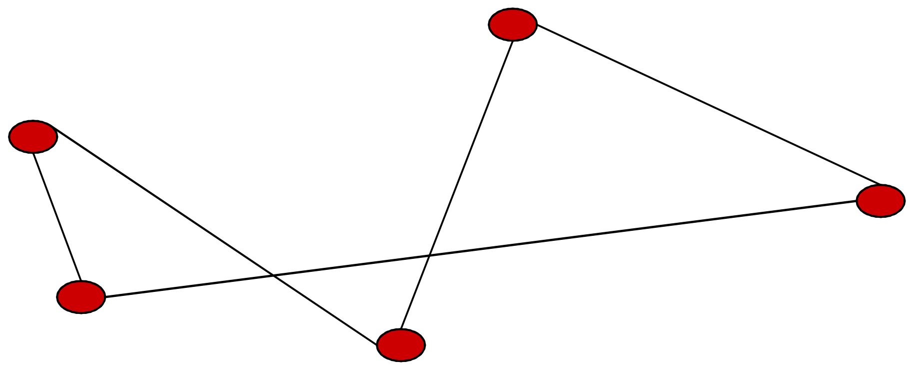
…

Boolean Satisfiability

Introduction

- Informal characterization
  - Combinatorial structure being optimized
  - There is a cost function to be optimized
    - At least we want to find a **good** solution
  - Searching all possible states is infeasible
  - No known algorithm for finding the solution efficiently
  - Some notion of similar states having similar costs

Example - TSP



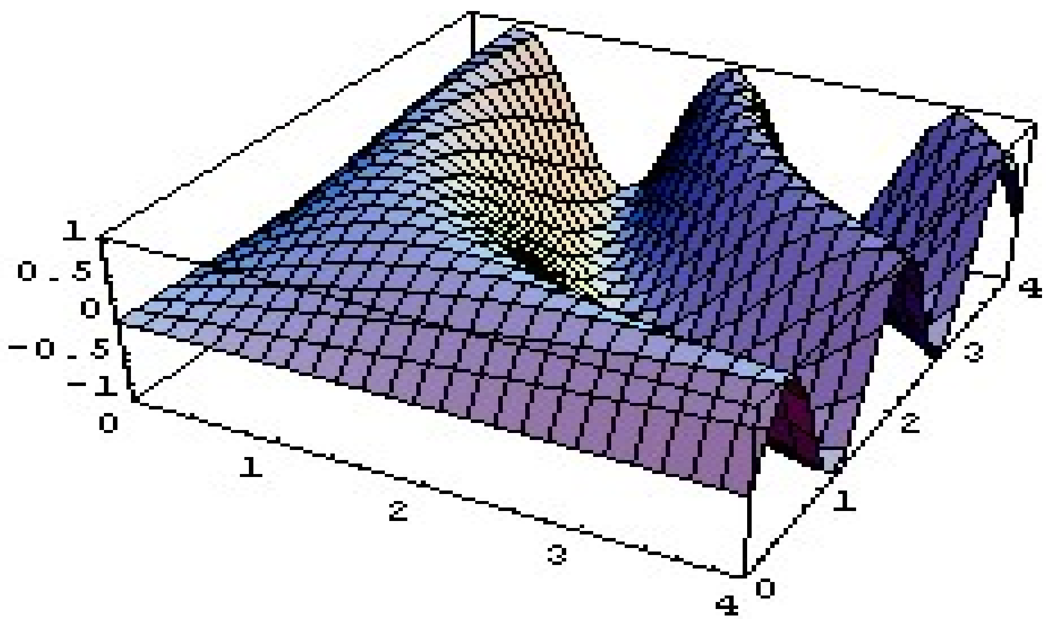
- Goal is to minimize the length of the route
- **Constructive method:**
  - Start from scratch and build up a solution
- **Iterative improvement method:**
  - Start with a solution and try to improve it

Constructive method

- For the optimal solution we could use A*!
  - But we do not really need to know how we got to the solution – we just want the solution
  - Can be very expensive to run

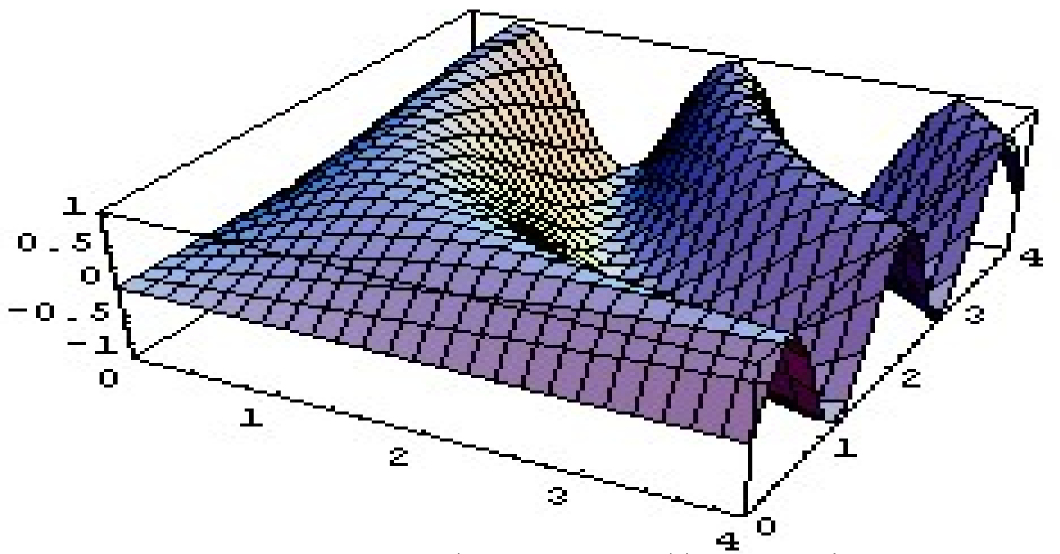
Iterative improvement methods

- Idea: Imagine all possible solutions laid out on a landscape
  - We want to find the highest (or lowest) point



Iterative improvement methods

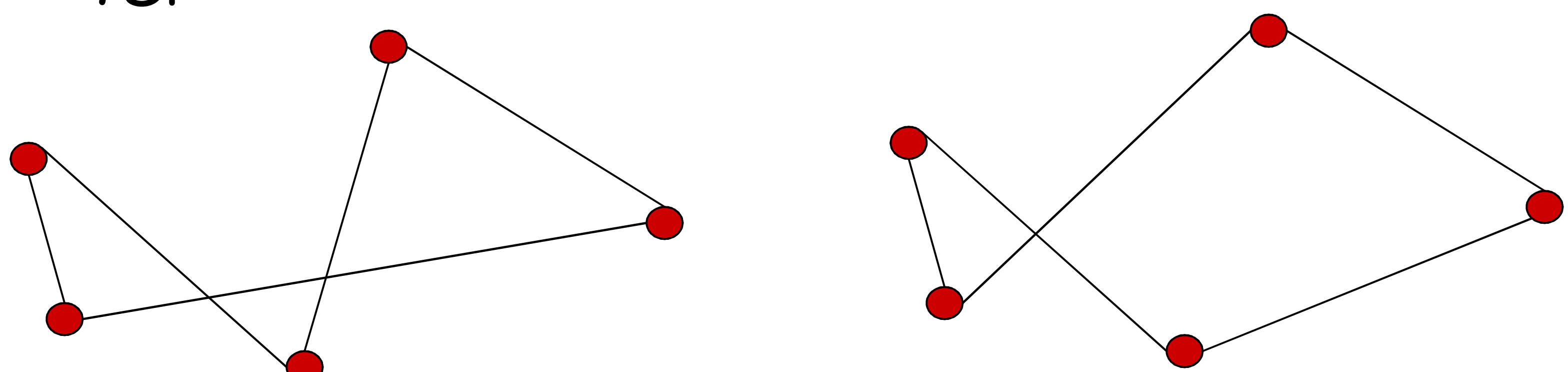
1. Start at some random point on the landscape
2. Generate all possible points to move to
3. Choose point of improvement and move to it
4. If you are stuck then restart



Iterative improvement methods

- What does it mean to "generate points to move to"
  - Sometimes called generating the **moveset**
- Depends on the application

TSP



2-swap

Hill-climbing

1. Start at some initial configuration S
2. Let $V = Eval(S)$
3. Let $N = MoveSet(S)$
4. For each $X_i \in N$

   Let $V_{max} = \max_i Eval(X_i)$

   and $X_{max} = argmax_i \; Eval(X_i)$
5. If $V_{max} \leq V$, return S
6. Let $S = X_{max}$ and $V = V_{max}$. Go to 3

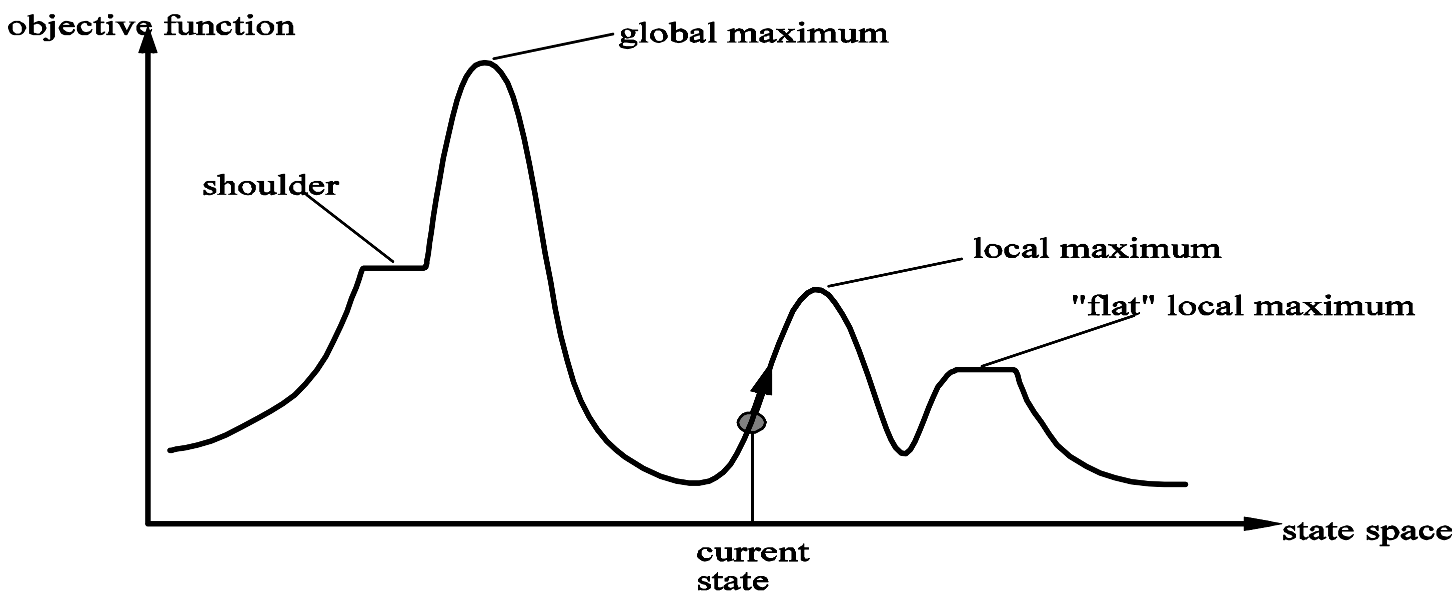
"Like trying to find the peak of Mt Everest in the fog", Russell and Norvig

Hill Climbing

- Always take a step in the direction that improves the current solution value the most
  - Greedy
- Good things about hill climbing
  - Easy to program!
  - Requires no memory of where we have been!
  - It is important to have a "good" set of moves
    - Not too many, not too few

Hill Climbing

- Issues with hill climbing
  - It can get stuck!
  - Local maximum (local minimum)
  - Plateaus



Improving on hill climbing

- Plateaus
  - Allow for sideways moves, but be careful since may move sideways forever!
- Local Maximum
  - Random restarts: "If at first you do not succeed, try, try again"
  - Random restarts works well in practice
- Randomized hill climbing
  - Like hill climbing except you choose a **random state from the move set**, and then move to it if it is better than current state. Continue until bored.

Hill climbing example: GSAT

A v ~B v C	1
~A v C v D	1
B v D v ~E	0
~C v ~D v ~E	1
~A v ~C v E	1

Configuration A=1, B=0, C=1, D=0, E=1

Goal is to maximize the number of satisfied clauses: Eval(config)=# satisfied clauses

GSAT Move_Set: Flip any 1 variable

WALKSAT (Randomized GSAT)

Pick a random unsatisfied clause;

Consider flipping each variable in the clause

- If any improve Eval, then accept the best
- If none improve Eval, then with prob p pick the move that is least bad; prob (1-p) pick a random one

Simulated Annealing

- Is hill climbing complete?
  - No: it never makes downhill moves
  - Can get stuck at local maxima (minima)
- Is a random walk complete?
  - Yes: it will eventually find a solution
  - But it is very inefficient

New Idea:

Allow the algorithm to make some "bad" moves in order to escape local maxima.

Simulated annealing

1. Let S be the initial configuration and $V = Eval(S)$
2. Let i be a random move from the moveset and let S_i be the next configuration, $V_i = Eval(S_i)$
3. If $V < V_i$ then $S = S_i$ and $V = V_i$
4. Else with probability p, $S = S_i$ and $V = V_i$
5. Goto 2 until you are bored

Simulated annealing

- How should we choose the probability of accepting a "bad" move?
  - Idea 1: $p = 0.1$ (or some other fixed value)?
  - Idea 2: Probability that decreases with time?
  - Idea 3: Probability that decreases with time and as $V - V_i$ increases?

Selecting moves in simulated annealing

- If new value V_i is better than old value V then definitely move to new solution
- If new value V_i is worse than old value V then move to new solution with probability

$$Exp(-(V - V_i)/T)$$

Boltzmann distribution: $T > 0$ is a parameter called temperature. It starts high and decreases over time towards 0

If T is close to 0 then the probability of making a bad move is almost 0

Properties of simulated annealing

- If T is decreased slowly enough then simulated annealing is guaranteed (in theory) to reach best solution
  - Annealing schedule is critical

- When T is high: Exploratory phase (random walk)
- When T is low: Exploitation phase (randomized hill climbing)

Genetic Algorithms

- Problems are encoded into a representation which allows certain operations to occur
  - Usually use a bit string
  - The representation is key – needs to be thought out carefully
- An encoded candidate solution is an individual
- Each individual has a fitness which is a numerical value associated with its quality of solution
- A population is a set of individuals
- Populations change over generations by applying operations to them

Typical genetic algorithm

- Initialize: Population P consists of N random individuals (bit strings)
- Evaluate: for each $x \in P$, compute $fitness(x)$
- Loop
  - For $i = 1$ to N do
    - Choose 2 parents each with probability proportional to fitness scores
    - **Crossover** the 2 parents to produce a new bit string (child)
    - With some small probability **mutate** child
    - Add child to the population
- Until some child is fit enough or you get bored
- Return the best child in the population according to fitness function

Crossover

- Consists of combining parts of individuals to create new individuals
- Choose a random crossover point
  - Cut the individuals there and swap the pieces

101|0101   011|1110

Cross over

011|0101   101|1110

Implementation: use a crossover mask m
Given two parents a and b the offsprings are
$(a \wedge m) \vee (b \wedge {\sim}m)$ and $(a \wedge {\sim}m) \vee (b \wedge m)$

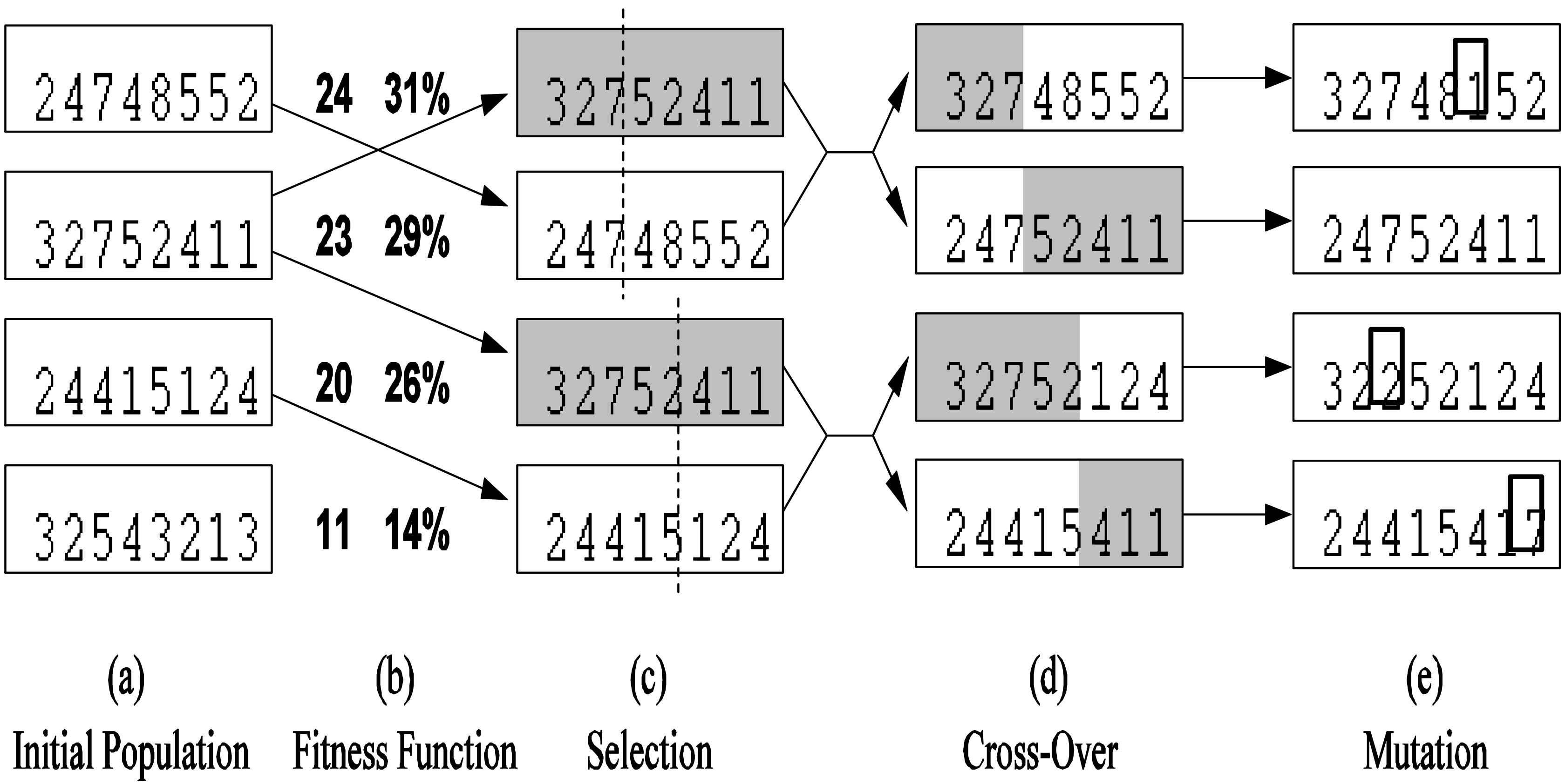
Mutation

- Mutation allows us to generate desirable features that are not present in the original population

- Typically mutation just means flipping a bit in the string

100111 mutates to 100101

Genetic Algorithms



Genetic algorithms and search

- Why are genetic algorithms a type of search?
  - States: possible solutions
  - Operators: mutation, crossover, selection
  - Parallel search: since several solutions are maintained in parallel
  - Hill-climbing on the fitness function
  - Mutation and crossover allow us to get out of local optima

Discussion of local search

- Useful for optimization problems!
- Often the second best way to solve a problem
  - If you can, use A* or linear programming or…
  - But local search is easy to program ☺
- Hill climbing always moves in the (locally) best direction
  - Can get stuck, but random restarts can be really effective
- Simulated annealing allows moves downhill