Advanced Algorithms

Floriano Zini
Free University of Bozen-Bolzano
Faculty of Computer Science

Academic Year 2013-2014

Lecture 9 – Network Optimization Algorithms (cont.)
Edmonds-Karp algorithm

- The Ford-Fulkerson algorithm does not specify which alternating augmenting path to use if there is more than one.
- In 1972, Edmonds and Karp proposed a heuristic for choosing the augmenting path.
  - choose the augmenting path with fewest edges.
- Using the heuristic, the path can be found in $O(|E|)$ time.
  - running a breadth-first search in the residual network.
  - $E$ is the set of edges.
- The algorithm ends after a polynomial number of iterations, independent of the edge capacities.

Edmonds-Karp algorithm – Pseudo code

1. create the residual network $Gr$
2. while there is some augmented path from $s$ to $t$ in $Gr$ do
   1. let $P$ be the path from $s$ to $t$ in $Gr$ with the fewest number of edges
   2. $c^* = \min_{(i,j) \in P} r_{ij}$
   3. send $c^*$ unit of flow along $P$
   4. update $r_{ij}$ for each $(i,j) \in P$

NB: $r_{ij}$ is a pair, including the residual capacities in both directions.
**Complexity: Ford-Fulkerson vs Edmonds-Karp**

Example of worst-case behavior of the Ford-Fulkerson algorithm
In each iteration, only a flow of 1 is sent across the network

A: source
D: sink

1 iteration

2 iterations

.... 2000 iterations

**Complexity: Ford-Fulkerson vs Edmonds-Karp**

Let’s apply Edmonds-Karp’s on the same example
The algorithm ends in 2 iterations

A: source
D: sink

The first iteration selects the path A-B-D (or A-C-D)
The flow is increased by 1000 units

The second iteration selects path A-C-D (or A-B-D)
The flow is increased by 1000 units
Edmonds-Karp algorithm – Example

Initial residual network $G$

- The residual capacity from $u$ to $v$ is $c_r(u,v) = c(u,v) - f(u,v)$, the total capacity, minus the flow that is already used.
- If the net flow from $u$ to $v$ is negative, it contributes to the residual capacity.

Edmonds-Karp algorithm – Example (cont.)

**Iteration 1**
- The shortest augmenting path $A \rightarrow D \rightarrow E \rightarrow G$ is selected.
- The residual capacity $c^*$ of the augmenting path is $c^* = \min\{3-0, 2-0, 1-0\} = 1$.
- The total flow is increased by 1.
- The new residual network is
Edmonds-Karp algorithm – Example (cont.)

**Iteration 2**
- the shortest augmenting path $A \rightarrow D \rightarrow F \rightarrow G$ is selected
- the residual capacity $c^*$ of the augmenting path is
  - $c^* = \min\{3-1, 6-0, 9-0\} = 2$
- The total flow is increased by 2
- The new residual network is

![Residual Network](image)

Edmonds-Karp algorithm – Example (cont.)

**Iteration 3**
- the shortest augmenting path $A \rightarrow B \rightarrow C \rightarrow D \rightarrow F \rightarrow G$ is selected
- the residual capacity $c^*$ of the augmenting path is
  - $c^* = \min\{3-0, 4-0, 1-0, 6-2, 9-2\} = 1$
- The total flow is increased by 1
- The new residual network is

![Residual Network](image)
Edmonds-Karp algorithm – Example (cont.)

**Iteration 4**
- the shortest augmenting path $A \rightarrow B \rightarrow C \rightarrow E \rightarrow D \rightarrow F \rightarrow G$ is selected
- the residual capacity $c^*$ of the augmenting path is
  - $c^* = \min\{3-1, 4-1, 2-0, 0-(-1), 6-3, 9-3\} = 1$
- The total flow is increased by 1
- The new (and final) residual network is

![Residual Network Diagram]

- The length of the augmenting path found by the algorithm (in red) never decreases
- The maximum flow is 5

Lecture 9 – Evolutionary algorithms

These slides are mainly taken from A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing
Randomized algorithms

- A deterministic algorithm behaves “predictably”
  - Given a particular input, it will always produce the same output through the same execution
- A randomized algorithm employs a degree of randomness as part of its logic
  - Given a particular input, its output is a random variable and also its execution cannot be predicted deterministically

Randomized algorithms (cont.)

- Randomized algorithms are often very simple and elegant, and their output is correct with high probability
- This success probability does not depend on the randomness of the input; it only depends on the random choices made by the algorithm itself
- There are two varieties of random algorithms
  - Monte Carlo algorithms always run fast but their output has a small chance of being incorrect
  - Las Vegas algorithms always output the right answer, but there is a small chance their running time is high
Consider the primality test algorithm presented in Lecture 3

```python
function primality(N):
    Input: Positive Integer N
    Output: yes/no
    Pick a positive integer a < N at random
    if \( a^{N-1} \equiv 1 \pmod{N} \):
        return yes
    else:
        return no
```

Is it a Monte Carlo or a Las Vegas random algorithm? Why?

It is a Monte Carlo Algorithm because it runs fast - \( O(n^2) \) - but it has a not null probability of error

Evolutionary Computing metaphor

- A **population** of individuals exists in an **environment** with **limited resources**
- **Competition** for those resources causes selection of those **fitter individuals** that are **better adapted** to the environment
- These individuals act as **seeds for the generation of new individuals** through recombination and mutation
- The **new individuals** have their fitness evaluated and **compete** (possibly also with parents) for **survival**
- Over time **natural selection** causes a **rise in the fitness of the population**
EC Metaphor

**EVOLUTION**

<table>
<thead>
<tr>
<th>Environment</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>Candidate Solution</td>
</tr>
<tr>
<td>Fitness</td>
<td>Quality</td>
</tr>
</tbody>
</table>

**PROBLEM SOLVING**

- Fitness → chance for survival and reproduction
- Quality → chance for seeding new solutions

Fitness in nature: **observed**, 2ndary
Fitness in EC: **primary**

Motivations for EC

- Developing, analyzing, applying problem solving methods a.k.a. algorithms is a central theme in mathematics and computer science
- Time for thorough problem analysis decreases
- Complexity of problems to be solved increases
- Consequence: ROBUST PROBLEM SOLVING technology needed
Problem type 1: Optimization

- We have a **model** of our system and **seek inputs** that give us a **specified goal**

```
? ———— Model ———— known ———— specified ———— Output
```

Optimization example 1: university timetabling

- Enormously big search space
- Timetables must be **feasible**
  - Vast majority of search space is infeasible
- Timetables must be **good**
  - Good is defined by a number of competing criteria
Problem types 2: Modelling

- We have corresponding sets of inputs & outputs and seek model that delivers correct output for every known input

![Diagram of model with inputs and outputs](image)

Modelling example: mushroom edibility

- **Classify** mushrooms as edible or not edible
- Evolving: **classifications models**
- **Fitness**: classification **accuracy** on training set of edible and not edible mushrooms

This is an example of **evolutionary machine learning**
Problem type 3: Simulation

- We have a given model and wish to know the outputs that arise under different input conditions

- Often used to answer “what-if” questions in evolving dynamic environments

Simulation example: evolving artificial societies

Simulating trade, economic competition, etc. to calibrate models

Use models to optimize strategies and policies
General schema of EAs

EA schema in pseudo-code

BEGIN

INITIALISE population with random candidate solutions;
EVALUATE each candidate;
REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
  1 SELECT parents;
  2 RECOMBINE pairs of parents;
  3 MUTATE the resulting offspring;
  4 EVALUATE new candidates;
  5 SELECT individuals for the next generation;
END
Main EA components

- Representation (encoding) of solutions into individuals
- Population
  - How big? How is it initialized?
- Evaluation
  - How to measure the goodness of individuals?
- Selection (parent selection, survivor selection)
  - How to pick up?
- Variation (mutation, recombination)
  - How to create new solutions?
- Termination condition
  - When the algorithm has found the best solution?

Representation

Phenotype space → Encoding (representation) → Genotype space

In order to find the global optimum, every feasible solution must be represented in genotype space.
Population

- **Role**: holds the candidate solutions of the problem as individuals (genotypes)
- Formally, a population is a **multiset** of **individuals**, i.e. repetitions are possible
- Population is the **basic unit of evolution**, i.e., the population is evolving, not the individuals
- **Selection** operators act on **population level**
- **Variation** operators act on **individual level**

Evaluation (fitness) function

- A.k.a. **quality** function or **objective** function
- **Role**:
  - Represents the **task** to solve, the **requirements** to adapt to (can be seen as "the environment")
  - **enables selection** (provides basis for comparison)
    - e.g., some phenotypic traits are advantageous, desirable, e.g. big ears cool better, these traits are rewarded by more offspring that will expectedly carry the same trait
  - Assigns a single **real-valued fitness** to each **phenotype** which forms the basis for selection
    - So the more discrimination (different values) the better
    - Typically we talk about **fitness being maximised**
      - Some problems may be best posed as minimisation problems, but conversion is trivial
Selection

Role:
- Identifies individuals
  - to become parents
  - to survive
- Pushes population towards higher fitness
- Usually probabilistic
  - high quality solutions more likely to be selected than low quality
  - but not guaranteed
  - even worst in current population usually has non-zero probability of being selected
- This stochastic nature can aid escape from local optima

Selection mechanism example

Example: **roulette wheel selection**

fitness(A) = 3
fitness(B) = 1
fitness(C) = 2

In principle, any selection mechanism can be used for parent selection as well as for survivor selection
Survivor selection

- A.k.a. replacement
- Most EAs use fixed population size so need a way of going from (parents + offspring) to next generation
- Often deterministic (while parent selection is usually stochastic)
  - Fitness based: e.g., rank parents+offspring and take best
  - Age based: make as many offspring as parents and delete all parents
- Sometimes a combination of stochastic and deterministic (elitism)

Variation operators

- Role: to generate new candidate solutions
- Usually divided into two types according to their arity (number of inputs):
  - Arity 1: mutation operators
  - Arity >1: recombination operators
    - Arity = 2 typically called crossover
    - Arity > 2 is formally possible, seldomly used in EC
- There has been much debate about relative importance of recombination and mutation
  - Nowadays most EAs use both
  - Variation operators must match the given representation
Mutation

- **Role**: causes small, random variance
- Acts on one genotype and delivers another
- Element of **randomness** is essential and differentiates it from other unary heuristic operators

**before**

```
1 1 1 1 1 1 1
```

**after**

```
1 1 1 0 1 1 1
```

Recombination

- **Role**: merges information from parents into offspring
- **Choice** of what information to merge is **stochastic**
- Most offspring may be worse, or the same as the parents
- Hope is that some are better by combining elements of genotypes that lead to good traits

![Recombination Diagram](image)
Initialisation / Termination

- **Initialisation** usually done at random
  - Need to ensure even spread and mixture of possible allele values
  - Can include existing solutions, or use problem-specific heuristics, to "seed" the population

- **Termination** condition checked every generation
  - Reaching some (known/hoped for) fitness
  - Reaching some maximum allowed number of generations
  - Reaching some minimum level of diversity
  - Reaching some specified number of generations without fitness improvement