Lecture 5: Genetic algorithms. Constraint Satisfaction

- Global search algorithms
  - Genetic algorithms
- What is a constraint satisfaction problem (CSP)
- Applying search to CSP
- Applying iterative improvement to CSP

Recall from last time: Optimization problems

- There is a cost function we are trying to optimize (e.g. travelling salesman problem)
- There may be constraints that need to be satisfied
- The state space is the set of all possible solutions, which is usually combinatorial
- Local search methods start with some initial solution and try to improve it iteratively by moving to “neighbouring” solutions.
  - Hill-climbing (aka gradient descent)
  - Simulated annealing
- Today: *global search*
  - Can jump around arbitrarily between possible solutions
  - Example: genetic algorithms, ant colony optimization etc.
Evolutionary computation

- Refers generally to computational procedures patterned after biological evolution
- Nature looks for the best individual (i.e. fittest)
- Many solutions (individuals) exist in parallel
- Evolutionary search procedures are also parallel, perturbing at random several potential solutions.

Genetic algorithms

- A candidate solution is called an individual
  - In a traveling salesman problem, an individual is a tour
- Each individual has a fitness: numerical value proportional to the evaluation function
- A set of individuals is called a population
- Populations change over generations, by applying operations to individuals: selection, mutation, crossover
- Individuals with higher fitness are more likely to survive, as well as to reproduce
- Individuals are typically represented by binary strings, to allow the evolutionary operations to be carried out easily
Mutation

- Mutation is a way of generating desirable features that are not present in the original population, by *injecting random changes*
- Typically mutation just means changing a 0 to a 1 (and vice versa)

![Mutation Illustration](image)

- The mutation rate $\mu$ gives the probability that a mutation will occur in an individual
- We can allow mutation in all individuals, or just in “offspring”

Crossover

- Consists of *combining parts of individuals* to create new individuals
- Single-point crossover: choose a crossover point, cut the individuals there, swap the pieces. E.g.:

  $\begin{array}{c}
  1011100 \\
  1001000 \\
  1111100 \\
\end{array}$

  $\begin{array}{c}
  1111000 \\
  1101100 \\
  1101100 \\
\end{array}$

- Implementation: use a crossover mask, $m$, which is a binary string. In our example, $m = 111000$.
  - Given two parents $i$ and $j$, the offspring are generated by: $(i \land m) \lor (j \land \neg m)$, and $(i \land \neg m) \lor (j \land m)$
- Multi-point crossover can simply be implemented using arbitrary (possibly random) masks
- In some applications, crossover has to be restricted, in order to produce “viable” offspring
Genetic algorithm generic code

GA(Fitness, threshold, p, µ, r)
1. Initialize population $P$ with $p$ random individuals
2. Repeat
   (a) For each $X_i \in P$, compute Fitness($X_i$)
   (b) If $\max_i \text{Fitness}(X_i) \geq$ threshold return the fittest individual;
   (c) Else generate a new generation $P_s$ through the following operations:
      i. Selection: Probabilistically select $(1 - r) \cdot p$ members of $P$ to “survive” and copy them to $P_s$
      ii. Crossover: Probabilistically select $r \cdot p/2$ pairs of individuals from $P$. For each pair, produce two offspring by applying the crossover operator (see next slides). Include all offspring in $P_s$.
      iii. Mutation: Randomly select $\mu \cdot p$ individuals and flip one randomly selected bit in each individual
     iv. $P \leftarrow P_s$

Selection: Survival of the fittest

- Like in natural evolution, we would like the fittest individual to be more likely to survive
- Several possible ways to implement this idea:
  - Fitness proportionate selection: $Pr(i) = \frac{\text{Fitness}(i)}{\sum_{j=1}^{p} \text{Fitness}(j)}$ (assuming fitness is positive)
  - Tournament selection: pick $i, j$ at random with uniform probability, then with probability $p$, select the fitter one
    Only requires comparing two individuals, which may be easier in some applications (e.g. games) than computing a fitness measure
  - Rank selection: sort all hypotheses by fitness; then probability of selection is proportional to rank
  - Softmax (Boltzman) selection:
    $$Pr(i) = \frac{\exp(Fitness(i)/T)}{\sum_{j=1}^{p} \exp(Fitness(j)/T)}$$
Elitism

- The best solution can "die" during evolution
- In order to prevent this, the best solution ever encountered can always be "preserved" on the side
- If the "genes" from the best solution should always be present in the population, it can also be copied in the next generation automatically, bypassing the selection process.
- Note that the best solution ever encountered is typically saved in hill climbing and simulated annealing as well

Genetic algorithms as search

- States: possible solutions
- Search operators: mutation, crossover, selection
- Parallel search, since several solutions are maintained in parallel
- An attempt at hill-climbing on the fitness function, but without following the gradient
- Mutation and crossover should allow getting out of local minima
- Very related to simulated annealing, but this is a global (not local) search method
TSP: Encoding as a GA

- Each individual is a tour (permutation of vertices)
- Mutation swaps a pair of edges (many other operations are possible, and have been tried in the literature)
- Crossover cuts the parents in two and swaps them \textit{if this does not create an invalid offspring}
- Fitness is the length of the tour
- Note that the GA operations (crossover and mutation) are a lot fancier for this realistic problem than for simple binary examples!

TSP example

$N = 13$

$P = 100$ elements in population

$\mu = 4\%$ mutation rate

$r = 50\%$ reproduction rate
TSP example: Initial generation

[Image of initial generation]

Best (lowest cost) element in population

Initial population

TSP example: Generation 15

[Image of generation 15]

Population at generation 15
The good and bad of GAs

- **Good things:**
  - Aesthetically *pleasing*, due to evolution analogy
  - If tuned right, *can be very effective* (good solutions found with fewer calls to the evaluation function than for simulated annealing)

- **Not-so-good things:**
  - Performance depends crucially on the encoding of the problem for the GA, and *good encodings are difficult to find*
  - *Many parameters to tweak!* Bad parameter settings can result in very slow progress, or the algorithm becoming stuck
  - Some quirky phenomena, e.g. *overcrowding*: too many individuals with the same genes are in the population, so genetic diversity is lost. Overcrowding occurs especially if the mutation rate $\mu$ is too low, or if multiple copies of the same individual can be kept in the next generation
Constraint satisfaction problems

- We want to find a solution that satisfies a set of constraints
  Eg. Sudoku, crossword puzzles
- Typically, very few “legal” solutions exist
- One can think of this problem as a cost function with minimum value at the solution, maximum value elsewhere
- Hence, optimization algorithms may not be easy to apply directly

Canonical example: Graph coloring

- Color the nodes such that two adjacent vertices are not the same color
- **Variables:** $V_i$
- **Domains:** Red, Blue, Green
- **Constraints:** If there is an edge between $V_i$ and $V_j$, their value (color) must be different
Constraint satisfaction problems (CSPs)

- A CSP is defined by:
  - A set of variables $V_i$ that can take values from domain $D_i$
  - A set of constraints specifying what combinations of values are allowed (for subsets of the variables)
  - Constraints can be represented:
    * Explicitly, as a list of allowable values (e.g., $C_1 = \text{red}$)
    * Implicitly, as a function testing for the satisfaction of the constraint (e.g., $C_1 \neq C_2$)
- A CSP solution is an assignment of values to variables such that all the constraints are true.
- We typically want to find any solution or find that there is no solution

Example: 4-Queens as a CSP

Put one queen in each column. In which row does each one go?

Variables $Q_1, Q_2, Q_3, Q_4$

Domains $D_i = \{1, 2, 3, 4\}$

Constraints:

- $Q_i \neq Q_j$ (cannot be in same row)
- $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

Translate each constraint into set of allowable values for its variables

E.g., values for $(Q_1, Q_2)$ are $(1, 3) (1, 4) (2, 4) (3, 1) (4, 1) (4, 2)$
Constraint graph

- **Binary CSP**: each constraint relates at most two variables
- **Constraint graph**: nodes are variables, arcs show constraints

- The structure of the graph can be exploited to provide problem solutions

Varieties of variables

- Boolean variables (e.g. satisfiability)
- Finite domain, discrete variables (e.g. colouring)
- Infinite domain, discrete variables (e.g. start/end of operation in scheduling)
- Continuous variables

Problems range from solvable in poly-time using linear programming to NP-complete to undecidable.
Varieties of constraints

- Unary: involve one variable and one value
- Binary
- Higher-order (involve 3 or more variables)
- Preferences (soft contraints): can be represented using costs, and lead to constrained optimization problems

Real-world CSPs

- Assignment problems (E.g., who teaches what class)
- Timetabling problems (E.g., which class is offered when and where?)
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Floor planning
- Puzzle solving (E.g. crosswords, Sudoku)
Applying standard search

- Assume a constructive approach:
  - States are defined by the values assigned so far
  - Initial state: all variables unassigned
  - Operators: assign a value to an unassigned variable
  - Goal test: all variables assigned, no constraints violated
- This is a general purpose algorithm, which works for all CSPs!

Example: Map coloring

Color a map so that no adjacent countries have the same color

Variables: Countries $C_i$
Domains: \{Red, Blue, Green\}
Constraints: $C_1 \neq C_2$, $C_1 \neq C_5$, etc.
Constraint graph:
**Standard search applied to map coloring**

Is this a practical approach? What is the complexity?

---

**Analysis of the simple approach**

- Maximum search depth = number of variables
  - All variables have to get some value
- Search algorithm to use: depth-first search
  - DFS is complete in this case because we know the maximum depth
- Branching factor = \( \sum_i |D_i| \) (at the top of the tree, at least)
  - *This can be a big search!*

But: this can be improved dramatically by noting the following:

- The order in which variables are assigned is irrelevant, so many paths are equivalent
- Adding assignments cannot correct a violated constraint
Backtracking search

• Like depth-first search but:
  – Fix the order of assignment (branching factor becomes $|D_i|$)
• Algorithm:
  – Select the next unassigned variable $X$
  – For each value $x_i \in D_X$
    * If the value satisfies the constraint, assign $X = x_i$ and exit the loop
  – If an assignment was found, continue with the next variable
  – If no assignment was found, go back to the preceding variable and try a different value for it.
• This is the basic uninformed algorithm for CSPs

Can solve $n$-queens for $n \approx 25$

---

Forward checking

Main idea: Keep track of legal values for unassigned variables

• When assigning a value for variable $X$
  – Look at each unassigned variable $Y$ connected to $X$ by a constraint
  – Delete from $Y$’s domain any value that is inconsistent with $X$’s assignment

Can solve $n$-queens up to $n \approx 30$
Heuristics for CSPs

More intelligent decisions on:

- which value to choose for each variable
- which variable to assign next

Given \( C_1 = \text{red}, C_2 = \text{green} \), choose \( C_3 \)
Choose \( C_3 = \text{green} \)

least-constraining-value

Now what variable next? Choose \( C_5 \):

most-constrained-variable

For ties: most constraining variable

Taking advantage of problem structure

- Worst-case complexity is \( d^n \) (where \( d \) is the number of possible values and \( n \) is the number of variables)
- But a lot of problems are much easier!
- Disjoint components - can be solved independently
- Tree-structured constraint graphs - \( O(nd^2) \)
- Nearly-tree structured graphs - Complexity \( O(d^c(n - c)d^2) \): Use cutset conditioning
  - Find a set of variables \( S \) which, when removed, turn the graph into a tree
  - Instantiate them all possible ways
  - Good if \( c \), the size of the cutset \( S \), is small
Iterative improvement algorithm for CSPs

- Start with a “broken” but complete assignment of values to variables
  - Allow states to have variable assignments that do not satisfy the constraints
- Randomly select conflicted variables
- Operators re-assign variable values
- This can be viewed as a relaxation of the cost function, which looks at the number of violated constraints as a cost to be minimized
  - Min-conflicts heuristic: choose value that violates the fewest constraints
  - I.e., approximate gradient descent on the total number of violated constraints
- Simulated annealing, genetic algorithms can be used here too.

Example: 4-Queens

- States: 4 queens in 4 columns \((4^4 = 256\) states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation function: number of attacks
Performance of min-conflicts

- Given random initial state, can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n=10^7$).
- The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio $R = \frac{\text{number of constraints}}{\text{number of variables}}$.

Summary

- CSPs are everywhere!
- Can be cast as search problems
- We can use either constructive methods or iterative improvement methods.
- Iterative improvement methods using min-conflicts heuristic are very general, and often work better.