Algoritmi si calculabilitate

lecture#7

Cluj-Napoca, 14.04.16

Agenda

- Greedy
- Dynamic programming
- Branch and Bound
- Search Techniques - Heuristics
- Genetic Algorithms (next course?)
  - History & Background
  - Search Space
  - Algorithm template
  - Parameters

Greedy (G)

- Indicated when an ordinary solution is required (although sometimes finds optimal solutions – particular-to-the-problem constraints to be added)
- Does not ensure optimality (but it may; problem-specific constraints)
- Still, could be used for solving optimal problems
- greedy technique for optimization problems (adding some extra-features and requirements)
- Optimality for greedy strategy need proof
- Very often used in practice for approximation algorithm
- For many NPC problems, very good approximations

Greedy (G) - contd.

- Search for global optimum based on local criteria
- Assumes that the local optimal choice leads to a global optimal solution
- You can't take it for granted. You need to

- Lots of important problems rely on the greedy strategy for finding the best solution

Greedy (G) - contd.

- In the search tree-space, the greedy approach picks the best on each level, and after each decision the "rest" of the tree is ignored
- if a problem cannot be solved with the greedy strategy, but we try to force it and run greedy on the problem, then:
  - it may either *not find any solution at all* (because it picks the local bests, moves further, and in the end realizes it was wrong somewhere down the road)
  - or give *a solution that is not globally optimal*

Greedy (G) - contd.

- Practical approach: in our everyday life
  - we may face "problems" that need to be solved in record time (or just timely)
  - and as resource-efficiently as possible
  - yet do not need a globally optimal solution. You just want it solved!
- rather solve it in reasonable time and find some (non-optimal) solution with reasonable supplementary resources than not having a solution at all
- Often for approximating solutions to NPC problems
- it is worth (if affortable) sacrificing optimality in favor of reducing overall complexity

Greedy (G) - contd.

C= candidate set contains all of the elements from which you can build a solution

select_best - procedure the *local best* candidate from C. It works on the basis of being "greedy," doesn't care about the global solution

feasible - procedure whether or not this locally optimal solution can contribute to the overall and global solution (does not behave greedily)

solution - procedure if we have found a solution

Greedy (G) - contd.

- A problem that can be solved with a greedy technique, almost always can be formulated in the following way:
  - begin with the choice of the local optimum
  - task reduced to another sub-problem of the same (or a related) type
  - recursively, the global optimal solution continues with the choice of a local optimum selection, and so on
  - the solution (which is optimal) is a collage of local optimum
- Running time always polynomial (most of the times linear in the input data size)

Greedy (G) - contd.

- Minimum Spanning Tree (MST)
  - Kruskal's – ordering edges; local optimum – least edge
  - Prim's – re-evaluating distances to partial built tree; local optimum – minimum distance to the tree
- Shorthest Path - Dijkstra's updating the closest vertex
- Knapsak Problem – 0/1 problem Has NO greedy optimal solution; fractional problem has an optimal greedy solution- local optimum – the most expensive unit item (normalized cost order)
- Coding – Huffman's codes – save 20-90% space
- Task scheduling (for unit time tasks, deadline and costs)
- Set cover – heuristic for approximating a solution for a NPC problem

Dynamic Programming (DP) – ex.

t(n)=O(n)

Only 2 memory cells (not an array, as it first considered)

Dynamic Programming (DP)

- Used to solve optimization problems
- Applicable whenever the problem exposes a recurrence relation
- DP breaks the large problem into incremental sub-problems, where each sub-problem can be solved and the solution is optimal
- Based on recurrence formula you can generate the final solution without altering the previously solved sub-problems or re-calculating some parts of the algorithm
-

Dynamic Programming (DP)– contd.

- based on the optimality principle
- a problem is said to exhibit *optimal structure* if the solution can be constructed efficiently from the previously calculated solutions of the sub-problems, which on their end are also optimal
- every part of an optimal solution is optimal.
- stores the solution of each sub-problem
- usually uses an array for this purpose (from a few variables, to multi-dimensional depending on how many parameters describe the solution(s))

Dynamic Programming (DP)– contd.

- the optimality principle is practically a mathematical recursive formula
- define how a solution of a sub-problem looks and how we can "*encode*" it (for storage)
- consider whether it is necessary to store everything; sometimes you can get away with less (a few variables instead of a whole array)
- filling the multi-dimensional array is performed using a *bottom-up* approach

Dynamic Programming (DP)– contd.

- When to choose DP?
  - *optimality principle* must apply (recursive formula)
  - the problem should *not* be solvable by the greedy strategy
  - lots of sub-problems are *identical* (if not, and sub-problems ARE DISJOINT, DI should be applied)

Dynamic Programming (DP)– contd.

- Ordering Matrix multiplication
  - Ex1: A=50x50; B=10x40; C=40x30; D=30x5
    - (A((BC)D) needs: BC =10x40x30=12000; (BC)D=12000+10x30x5=13500; (A((BC)D)=13500+50x10x5=
    - (A(B(CD))) needs: CD=40x30x5=6000; B(CD)=10x40x5+6000 = 8000; (A(B(CD))) = 8000+50x10x5 =
  - Ex2: A=1x100; B=100x2; C=2x50
    - A(BC) needs BC=100x2x50=10^4; A(BC)= 10^4+1x100x50= ;
    - (AB)C needs: AB=2x10^2; (AB)C =2x10^2+1x2x50= ;
- Longest sub-pattern within a pattern
- Optimal decomposition of a polygon in triangles (has a similar solution with Matrix multiplication)
- All-Pairs shortest path- Floyd's
- Shortest distance in a graph - Bellman-Ford's

Branch and Bound (B&B)

- Injects some intelligence into the naïve but complex breadth-first search
- Nothing injected => pure BFS => all branches completely generated
- "everything" injected (that is, enough information available, and injected to the strategy) => greedy => just one branch generated
- Can be viewed as an "intelligent" (or informed) version of bfs in the search space
- Changes the searching through the entire search space by adding some criteria, according to which the complexity of the BFS can be reduced
- Costs are associated to searched path

Branch and Bound (B&B) – contd.

- Costs – related to how far is a node located from the initial node and how close to the solution node;
- a node is a more viable candidate toward the solution if its cost is less than he costs of the other nodes
- add an essence of to the meaning
- Regular Q from BFS becomes a
- Nodes are added to the Q according to their associated costs

Branch and Bound (B&B) – contd.

- actually all nodes remain in the Q
- we haven't lost the not-so-possible candidates (i.e. candidates nodes with "poor" cost values remain at the end of the Q, to be chosen in case the "better" ones do not lead to a solution)
- they are stored at the end of the priority Q, so the technique doesn't neglect the rest of the possible options (just postpones the less reliable)

Branch and Bound (B&B) – contd.

- B&B = two main actions (with a possible enhancement of another step)
  - *branching* defines the tree structure from the set of candidates in a recursive manner;
  - *bounding* - calculates the upper and lower bounds of each node from the tree (tree= search space structure)
  - *pruning* - (optional) if the lower bound for some node of the tree is greater than the upper bound of some other node of the tree, then the first node of the tree can be "discarded" from the search (i.e. an new better option replaces an existing option).

Branch and Bound (B&B) – contd.

- Similar to BT; the differences are:
  - B&B is used only in case of optimization problems while BT in all types
  - B&B doesn't limit us to a particular way of traversing the states graph (although deals with BFS + cost-preferences of so-far-most-efficient branch which is kind of dfs), while BT uses DFS with preorder
- Similar to BFS; the differences are:
  - Q=priority Q
  - the highest priority element (most promising branch at the moment) is always on the first position in the Q

Branch and Bound (B&B) – contd.

- How are priorities calculated?
- The cost f(x)=g(x)+h(x) estimates the cost of the solution on a path from the start point (root of the search tree) to the final point (goal node) via node x.
- g(x) exact value – calculated cost of the path from Start to x
- h(x) estimated value – approximation of the cot of the path from x to Goal; The efficiency of the B&B relies on this function.
- The better approximation, less "expansive" searching
- The technique works good enough in many cases with rough approximations (including the worst, i.e. h(x)=0)

Branch and Bound (B&B) – contd.

- The technique works with 2 lists:
  - Candidate list (the priority Q) – contains the nodes (i.e. states) to be evaluated (generalization of the boundary)
  - Expanded list (containing the configurations that have been passed through) – contains the nodes already evaluated
- A solution is found only if the h(x) returns 0 (thus, a solution leaf) the priority Q becomes empty.

Branch and Bound (B&B) – contd.

B&B

Branch and Bound (B&B) – contd.

- B&B may also be a base of various heuristics (defined by the bound criterion)
- stop branching when the gap between the upper and lower bounds becomes smaller than a certain threshold - used when the solution is "good enough for practical purposes" and can greatly reduce the computations required.
- Such cases are applicable when f(x) is *noisy* or is the result of statistical estimates
- application often in biology when evaluating evolutionary relationships, where the data sets are often impractically large (therefore no GT or BT could be applied).

Branch and Bound (B&B) – contd.

- combinatorial optimization problems
- Knapsack problem
- Traveling salesman problem (TSP)
- Integer programming
- Nonlinear programming

History of GAs

- Part of evolutionary computing (IA field)
- evolutionary computing introduced in the 1960s by I. Rechenberg in his work "Evolution strategies"
- Genetic Algorithms (GAs) -John Holland book "Adaption in Natural and Artificial Systems" 1975
- Genetic Programming (GP) John Koza - 1992 has used genetic algorithms to evolve programs

Biological Background

- Cells = set of chromosomes
- Chromosomes = strings of DNA
- DNA substrings = genes
- gene encodes a particular protein (responsible for some particularity of the individual)
- Genome = complete set of genetic material (all chromosomes)
- Genotype = particular set of genes in genome
- Phenotype = development of the genotype after birth (an observable characteristic or trait (feature) of an organism - property or behavior)

Biological Background – contd.

- Reproduction - recombination (or crossover) = genes from parents form the new chromosome
- The new created offspring can then be mutated
- Mutation = a mutation is a change of the nucleotide sequence of the genome as a result from unrepaired damage to DNA
- the elements of DNA are a bit changed (mainly caused by errors in copying genes from parents in biology; on purpose in GAs).
- fitness = success of the organism in its life (in biology; an efficiency measure for GAs)

Search Space

- search space (state space) = the space of all feasible solutions (objects among which the desired solution is)
- Each point in the search space represents one feasible solution
- Each feasible solution can be "marked" by its value or fitness for the problem
- Solution = one point (or many) among the feasible solutions, i.e. one point in the search space.
- looking for a solution = looking for some extreme in the search space
- Characteristics of the search space:
  - can be known by the time of solving a problem,
  - usually we know only a few points from it and we are generating (generating them might be costly – monetary, in time, as physical pain, …) other points as the process of finding solution continues.

Problem complexity

- Space very large (sometimes unbounded!)
- Identify a good strategy to approach the optimal point
- Strategies: greedy, hill climbing, tabu search, simulated annealing, genetic algorithms.
- NP-complete problems:
  - Check a solution in poli time
  - Unknown poli alg to solve the problem
  - Nondeterministic machines may find solution
  - Guess a "good" solution?

Already discussed solutions

- Greedy
- Bfs
- Dfs
- Dynamic programming
- B&B
- Advantages and limitations

Uniform cost search (A*)

- Hart, Nilsson, Raphael (68) as enhancement of Dijkstra's (59).
- Dijkstra's = A*, with no heuristic (h=0).
- Greedy = A*, with no prior (g=0) and 1 single element in the queue (end local estimation of h, of 1 step only)
- B&B = A*, with simple h
- uses a best-first (choice of continuation triggered by an efficiency estimate) search to find the least-cost path from an initial node to a goal node

Uniform cost search (A*)

- Searches paths that *seem* more likely to lead towards goal
- Evaluates a performance of the choice as performance function: *f(n)=g(n)+h(n)*
  - *f(n)* represents the estimated cost on the cheapest path from the start node to the goal
  - *g(n)* as an exact measure of the path from the start node to the current node (as for *uniform cost search*),
  - *h(n)* estimation of the remainder path to the goal (as for *greedy search*) – heuristic (by "guess"). The heuristic must be admissible (not overestimate h)

Uniform cost search (A*)

- Maintains a (priority) queue of alternate paths
- the node with the lowest f value is extracted from the queue
- its neighbors are updated (f and h) accordingly and added to the queue.
- the algorithm continues until a goal node has a lower value than any node in the queue (or until the queue is empty)
- The f value of the goal is the length of the shortest path (at the goal h=0).
- From the actual shortest path, the algorithm may update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node.
- the method is both *complete* (if a solution exists, A* finds it) *and optimal* (no optimal algorithm employing the same heuristic will expand fewer nodes than A*) (Russell & Norvig, 95)

Enhancements

- Iterative improvement techniques for boosting search strategies.
- hill climbing
  - For optimum solutions
  - makes changes to improve the current state.
  - does not maintain a search tree
  - it moves in the direction of increasing value
  - Efficient in practice (often used); yet it suffers the drawback of becoming trapped in local optima.
- simulated annealing
  - for optimization problems
  - allows escaping a local optimum, by taking some steps to break out.
  - is an effective strategy for a good approximation of the global optimum in a large search space

Hill Climbing

- Optimization technique
- Performs local search
  - starts with a random solution
  - iteratively makes small changes to the solution, each time improving it a little.
  - When no improvement obtained, it terminates.
  - Ideally, the current solution is close to optimal, but it is not guaranteed (that will ever come close to the optimal solution)

simulated annealing

- *Name*: technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects
- Probabilistic meta-heuristic for global optimization problems
- locating a good approximation to the global optimum in a large search space
- often used when the search space is discrete
- Parameters:
  - the state space (diameter of the search graph)
  - the goal function (evaluation of the performance)
  - the candidate generator procedure (based on transition probabilities)
  - the acceptance probability function (chooses the next state)

Tabu Search

- Glover (86)
- Combinatorial Optimization; local search
- It is a metaheuristic
- enhances the performance of local search by using memory structures (usually lists): once a potential solution has been determined, it is marked as "taboo”
- modifies the neighborhood structure of each solution as the search progresses; the new neighborhood, are determined through the use of memory structures

Basic Genetic Algorithm

Generate random population of *n* chromosomes (initial solutions of the problem; simply guess)

Evaluate the fitness *f(x)* of each chromosome *x* in the population (the "goodness" of each individual - potential solution)

Build a new population = repeating following steps (the process in the next slide) until the termination condition is met

…

if the termination condition is satisfied,

then , and return the best solution (individual) in the current population

else repeat the loop

Basic Genetic Algorithm (cont)

- steps for building it

Select two parent chromosomes (individuals) from a population (the current one) according to their fitness (the better fitness, the bigger chance to be selected)

With a crossover probability cross over the parents to form (a/two) new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

With a mutation probability mutate new offspring at each locus (position in chromosome).

Place new offspring in a new population

Use new generated population for a further run of algorithm

Termination

- A solution is found that satisfies minimum criteria (i.e. an individual in the current population matches the condition)
- of generations reached
- (computation time/money) reached
- The highest ranking solution's fitness is reaching or has reached a such that successive iterations no longer produce better results (i.e. the fitness increase of the whole population *or* of the best individuals is below a threshold – too small further improvement) – learning curves
- Manual (in the research faze)
- of the above (in most cases fixed number of generation or given value/threshold of fitness, whichever occurs first)

Evaluation

- The quality of the solution depends on:
  - The initial population (size and how "close" the initial chromosomes are to the best solution)
  - Operators of GA
    - Crossover (type and quantity)
    - Mutation (quantity)
  - Fitness function (how "accurate" it measures the quality of the solution)
  - Proper values/types determined via learning curves
- The ability to reach a good solution (even the convergence) is strongly related on:
  - Codification
  - Fitness function
  - Expert knowledge ability and expertise

Chromosomes Encoding and Operator

- Usually as binary strings; each chromosome (individual in the population) is a binary string
- Crossover: selects genes from parents and creates a new offspring
  - Single point crossover: choose randomly some crossover point and genes are selected complementary from parents

Chromosome1	11011 | 00100110110
Chromosome2	11011 | 11000011110
Offspring1	11011 | 11000011110
Offspring2	11011 | 00100110110

  - multiple points crossover: multiplies the above strategy in many points
  - research suggests more than two "parents" are better to be used to reproduce a good quality chromosome.
- Specific crossover (dedicated to specific problem) can improve performance of the genetic algorithm
- Generally the average fitness will have increased by this procedure for the population (ensures the convergence of the GA)

Mutation

- Performed to avoid local optima (like in hill climing)
- "forces" some changes in the offsprings encoding
- preventing the population from becoming too similar to each other, thus slowing (time) or even stopping (not converge) evolution
- For binary encoding some (randomly chosen) bits are flipped (0/1)

Original offspring 1 1101111000011110
Mutated offspring 1 1100111000011110
Original offspring 2 1101100100110110
Mutated offspring 2 1101101100110110

Parameters

- Crossover probability (rate) - how often will be crossover performed.
  - Goal: increase the quality of individuals in the set
  - 0% (no crossover) - offspring is exact copy of parents.
  - X% (is a crossover)- x% offspring are made from parts of parents' chromosome.
  - 100% -all offsprings are made by crossover
- Mutation probability – the percentage of the chromosome that is mutated
  - Goal: "change the blood"
  - 0% - not at all (trapped in local minimum)
  - 100% - all bits flipped (chaotic move in space)
  - As mutation is performed to avoid local optimum, abuse will hurt (approaches to random search)

Parameters

- Population size = dimension of one generation
  - too few => few possibilities to perform crossover, so a small part of the search space is explored (i.e. solution elsewhere)
  - too many => GA slows down (i.e. search everywhere, approaches to exhaustive search)

Recommended values

- Crossover rate - about 80%-95% (sometimes 60% suffice)
- Mutation rate - very low; Best rates reported are about 0.5%-1% (sometimes even 0.2%)
- Population size - most of the cases 20 – 30 (sometimes 50 – 100)
- Crossover and mutation type – depends on the problem and on the encoding
- Other: selection and encoding (next)

Selection

- Defines how parents (individuals from the current population) are selected for breeding
- According to Darwin's evolution theory the best ones survive and create new offspring
- Techniques:
  - Roulette Wheel Selection
  - Rank Selection
  - Steady-State Selection
  - Elitism

Roulette Wheel Selection ( )

- Breeding parents are selected according to their fitness values (i.e. the better the chromosomes, more chances to be selected)
- where are placed all chromosomes in the population, the dimension on the wheel according to its fitness function
- the fitness level is used to associate a probability of selection with each individual chromosome
- Chromosome with bigger fitness are selected more times (Darwin's law)
  - Calculate fitness sum ( ) = summation of the fitness function of all individuals in the population of all chromosome
  - Probability for an individual to be selected is f_i/S, where f_i is i^{th} fitness function
- Drawback: in case fitness differs very much. If the best chromosome fitness is 90% in the whole (initial) set, the other chromosomes have very few chances to be selected.

Rank Selection

- Rank each individual in the population
- Selection probability is reversed proportional to the rank (better individuals, larger probability)
- All chromosomes have a chance to be selected
- Overcome the drawback of roulete-weel
- While keeping a performance criterion

Tournament Selection http://www.it-weise.de/projects/book.pdf

- k (often k=2) elements are picked from the population (with replacement, i.e. good chances to be selected again) and compared with each other in a tournament.
- The winner of this competition will then enter mating pool
- individual will, on average, participate in two tournaments (for k=2).
- The solution candidate of the population will win all the contests it takes part in and thus, again on average, contributes approximately two copies to the mating pool.
- The individual of the population is better than 50% of its challengers but will also loose against 50%. It will enter the mating pool roughly one time on average.
- The individual in the population will lose all its challenges to other solution candidates and can only score even if competing against itself. It will not be able to reproduce in the average case.

Steady-State Selection
Elitism

- not particular methods of selecting parents
- Ensures surviving of individuals for the next generation
- Steady-State Selection
  - good chromosomes for creating a new offspring
  - bad chromosomes are removed & the new offspring is placed in their place
  - the rest of population survives to new generation.
- Elitism
  - copies the best chromosome (or a few best chromosomes) to the new population
  - the rest is done in classical way
  - can increase performance of GA; it prevents losing the best found solution so far (i.e. keeps searching around local optimum, in case it represents global optimum)

Encoding

- Depends on the problem
- Binary Encoding
  - every chromosome is a string of bits, 0 or 1.
  - Gray code, binary numeral system where two successive values differ in only one bit (therefore, a mutation – new genotype, does not produce major changes in the phenotype)
- Permutation Encoding
  - in ordering problems (traveling salesman, task ordering)
  - Ex: order of cities
- Value Encoding
  - chromosome is a string of some values
  - Values can be anything connected to problem (form numbers to objects)
  - Requires special crossover and mutation
- Tree Encoding
  - for evolving programs or expressions, for genetic programming