Advanced Algorithm Design and Analysis (Lecture 14)

SW5 fall 2004

Simonas Šaltenis

E1-215b

simas@cs.aau.dk

Plan for Lecture 15

- Group presentations, no more than 20 minutes each + 5-10 minutes of questions, discussion.
- Questions to address:
  - What is the problem?
    - Input, output
    - What interface are you implementing?
  - What are the possible algorithmic solutions?
    - Description:
      - Data structures used
      - Algorithm design techniques used
    - Theoretical comparison:
      - Worst-case running time?
      - Amortized running time
      - Space used

Plan for lecture 15

- Experiments
  - Settings, data sets
  - Average running time
  - Reflection (why the results are as they are? Is this as expected?)
- What are the implementation issues?

Backtracking, Branch&Bound

- Main goals of the lecture:
  - *to understand the principles of **backtracking** and **branch-and-bound** algorithm design techniques;*
  - *to understand how these algorithm design techniques are applied to the example problems (**CNF-Sat**, **TSP**, and **Knapsack**).*

Coping with NP-completeness

- Options for coping with an NP-complete problem:
  - We may be able to find provably **near-optimal** solutions in polynomial time – approximation algorithms
  - *Special cases* maybe solvable in polynomial time
  - Just use an exponential algorithm – either hope that the input is *very small* or that the worst case manifests itself very rarely
    - use different **heuristics** to speed up search through the space of possible solutions

Propositional logic

- George Boole (1815-1864) – reduced popositional logic to algebraic manipulations
  - A propositional logic formula is composed from:
    - Boolean variables (*x*, *y*, …) – can get values *true*(1) and *false*(0)
    - Boolean operators:
      - Negation "Not" (notation: $\overline{x}$)
      - Conjunction "And" (*notation*: $x \cdot y$)
      - Disjunction "Or" (*notation*: $x+y$)
  - Example: $(\overline{r} + w) \cdot (m + f)$
  - **Satisfiability**: Give an assignments of values to variables, if there is one, that makes the input formula true (1)

Map labeling

- *Why do we need satisfiability*?
  - Modeling different problems with propositional logic formulas
  - **Map labeling**:
    - Four positions for a label of a city: {above-right, above-left, below-right, below-left}
    - Goal: find a labeling where city names in a map do not overlap

Map labeling

- *What are the variables and how do we specify constraints (conflicts) as a formula?*
  - Each city *x* two variables:
    - x_a: label is above if 1, else label is below
    - x_r: label is right if 1, else label is left
  - Describe each constraint:
    - For example: $\overline{(o_a \cdot \overline{f_a} \cdot f_r)} = (\overline{o_a} + f_a + \overline{f_r})$
    - Connect constraints by "and"

CNF

- Conjunctive Normal Form (CNF) for boolean formulas
  - CNF is a conjunction of **clauses**
  - Each clause is a disjunction of **literals**
  - Each literal is a variable or its negation.
  - Any boolean formula can be transformed to CNF:
    - For example: $(\overline{o}_a + f_a + f_r)(k_r + \overline{k}_a + \overline{e}_r + \overline{e}_a)(k_r + k_a + \overline{e}_r + e_a)$
  - *When is CNF satisfied*?

CNF-Sat brute force

- CNF-Sat is NP-complete
- *How do we solve it then with brute force?*
  - Consider all possible assignments of truth values to all variables in the formula:

x_1	x_2	…	x_n	*formula*
0	0	…	0	
0	0	…	1	
…	…	…	…	
1	1		1	

  - *What is the running-time*?

Structure of the NPC problem

- We can do better in practice:
  - We use the structure of an NP-complete problem:
    - If we have a certificate, we can check
    - A certificate is constructed by making a number of choices
    - *What are these choices for the CNF-Sat*?
  - **Configuration** (*X*, *Y*):
    - *Y* – choices made so far (part of the certificate)
    - *X* – a subproblem remaining to be solved

Backtracking

- **Backtracking** algorithm design technique:
  - Have a ***frontier*** set of configurations .
  - *Observation* 1: sometimes we can see that configuration is a ***dead end*** – it can not lead to a solution
    - we *backtrack*, discarding this configuration
  - *Observation* 2: If we have several configurations, some of them may be more "promising" than the others
    - We consider them first

Backtracking

```
Backtracing(P)       // Input: problem P
F ← {(P, ∅)}         // Frontier set of configurations
while F ≠ ∅ do
  Let (X,Y)∈F - the most "promising" configuration
  Expand (X,Y), by making a choice(es)
  Let (X_1,Y_1), (X_1,Y_1), ..., (X_k,Y_k) be new configurations
  for each new configuration (X_i,Y_i) do
    "Check" (X_i,Y_i)
    if "solution found" then
      return the solution derived from (X_i,Y_i)
    if not "dead end" then
      F ← F ∪ {(X_i,Y_i)}   // else "backtrack"
return "no solution"
```

Details to fill in

- Important "details" in a backtracking algorithm:
  - *What is a configuration (choices and subproblems)*?
  - *How do you select the most "promising" configuration from F*? – Ordering search
    - Traditional backtracing uses LIFO (stack)– depth-first search, one could use FIFO (queue) – breadth-first search, or some more clever ***heuristic***
  - *How do you extend a configuration into subproblem configurations*?
  - *How do you recognize a dead end or a solution*?

CNF-Sat: Promising configuration

- *CNF-Sat*: *What is a configuration*?
  - An assignment to a subset of variables
  - CNF with the remaining variables
- *What is a promising configuration?*
  - Formula with the smallest clause
    - *Idea*: to show as soon as possible that this is a dead end
  - Other choices are possible
- *How do we generate subproblems*?
  - Take the smallest clause and pick a variable *x*:
    - One subproblem corresponds to x = 0
    - Another to x = 1

CNF-Sat: generating subproblems

- Generating subproblems*:*
  - For each choice of assignment to *x* do:
    - 1. Assign the value to *x* everywhere in the formula
      - If a literal = 1, the clause disapears,
      - If a literal = 0, the literal disapears
    - 2. If this results in a clause with single literal, assign 1 to that literal and propagate as in 1.
    - Do 2. while there are clauses with single literal
  - *How do we recognize a dead-end or a solution*?
    - Dead-end: single-literal clause is forced to be 0
    - Solution: all clauses disappear

Example

- This is a so-called David-Putnam procedure
  - Do the example:

$$(\overline{x_1} + x_2 + \overline{x_3}) \cdot (\overline{x_2} + x_3 + x_4) \cdot (x_1 + x_2 + \overline{x_4})$$

  - *What is the running time of this algorithm*?

Optimization problems

- *Can we use a backtracking algorithm to solve an optimization problem (not a decision problem)*?
  - For example: In TSP problem we need to find a *shortest* hamiltonian cycle, not just some hamiltonian cycle
  - *Idea*: Use a backtracking algorithm but modify it so that when a solution *S* is found:
    - If *S* is better than the best solution seen so far (*B*), update *B*=*S*, otherwise discard solution.
    - Continue

Pruning

- This works, but we can do better – discard solutions earlier:
  - If we can estimate the lower-bound *lb* on the cost of a solution derived from a configuration *C,* then we can discard *C*, whenever *lb*(*C*) is larger than the cheapest solution found so far (*B*)
  - This is called ***pruning***:
    - For example, if a partially constructed path *P* in TSP problem is longer than the best solution found so far, we can discard *P*
- Backtracking together with pruning constitute the ***branch-and-bound*** algorithm design technique

Branch-and-Bound algorithm

```
Branch-and-Bound(P) // Input: minimization problem P
F ← {(P, ∅)}       // Frontier set of configurations
B ← (+∞, ∅)        // Best cost and solution
while F ≠ ∅ do
  Let (X,Y)∈F - the most "promising" configuration
  Expand (X,Y), by making a choice(es)
  Let (X_1,Y_1), (X_1,Y_1), ..., (X_k,Y_k) be new configurations
  for each new configuration (X_i,Y_i) do
    "Check" (X_i,Y_i)
    if "solution found" then
      if the cost c of (X_i,Y_i) is less than B cost then
        B ← (c,(X_i,Y_i))
      else discard the configuration (X_i,Y_i)
    if not "dead end" then
      if lb(X_i,Y_i) is less than B cost then  // pruning
        F ← F ∪ {(X_i,Y_i)}   // else "backtrack"
return B
```

TSP: Branch-and-Bound

- Let's solve TSP with branch-and-bound:
  - Let's start by assuming edge $e=(v,w)$ is in a tour
  - Then the problem is: to find a shortest tour visiting all vertices starting from v and finishing in w in the graph $G=(V,E-\{e\})$
  - *What is a configuration*?
    - Path P constructed so far
    - Remaining subproblem: $G=(V-\{\text{vertices in } P\},E-\{e\})$
  - *How do I generate new configurations?*
  - *Which may be chosen as the most promising?*

TSP: Branch-and-Bound

- *When do we see that a path is a dead-end*?
  - A partial path P is a dead-end, if $G=(V-\{\text{vertices in } P\}, E-\{e\})$ is disconnected
- *How do we define a lower bound function for pruning*?
  - The lower bound on the cost of the tour can be the cost of all edges on P plus $c(e)$
- When we are done with an edge e, we can repeat the same for the remaining edges
  - B, the cheapest tour seen so far, does not have to be reset for each starting edge – improved pruning

Example

- Run the branch-and-bound algorithm to find TSP on the following graph: