Heuristics, Closing Thoughts

Lecture 25

May 3, 2011

Part I

Heuristics

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out
  - use massive computational power
- Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is OK.
- Settle for sub-optimal (aka approximate) solutions, especially for optimization problems

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

- Consider special cases of the problem which may be tractable.
- Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
  - stop algorithm when time/resources run out
  - use massive computational power
- Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is OK.
- Settle for sub-optimal (aka approximate) solutions, especially for optimization problems

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

NP $\subseteq$ EXP.

Proof.

Let $X \in NP$ with certifier C. To prove $X \in EXP$, here is an algorithm for X. Given input s,

- For every t, with $|t| \leq p(|s|)$ run $C(s, t)$; answer "yes" if any one of these calls returns "yes", otherwise say "no". □

Every problem in **NP** has a brute-force "try all possibilities" algorithm that runs in exponential time.

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier **C**. To prove $X \in$ **EXP**, here is an algorithm for **X**. Given input **s**,

- For every **t**, with $|t| \leq p(|s|)$ run $C(s, t)$; answer "yes" if any one of these calls returns "yes", otherwise say "no". □

Every problem in **NP** has a brute-force "try all possibilities" algorithm that runs in exponential time.

Examples

- **SAT**: try all possible truth assignment to variables.
- **Independent set**: try all possible subsets of vertices.
- **Vertex cover**: try all possible subsets of vertices.

Improving brute-force via intelligent backtracking

- Backtrack search: enumeration with bells and whistles to "heuristically" cut down search space.
- Works quite well in practice for several problems, especially for small enough problem sizes.

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. φ' is CNF formula obtained by setting $x_i = 0$ and simplifying
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
   - If "not sure" recursively solve φ'
   - If φ' is satisfiable, return "yes"
4. φ'' is CNF formula obtained by setting $x_i = 1$
5. Run simple check on φ'': returns "yes", "no" or "not sure"
   - If "not sure" recursively solve φ''
   - If φ'' is satisfiable, return "yes"
6. Return "no"

Certain part of the search space is pruned.

Example

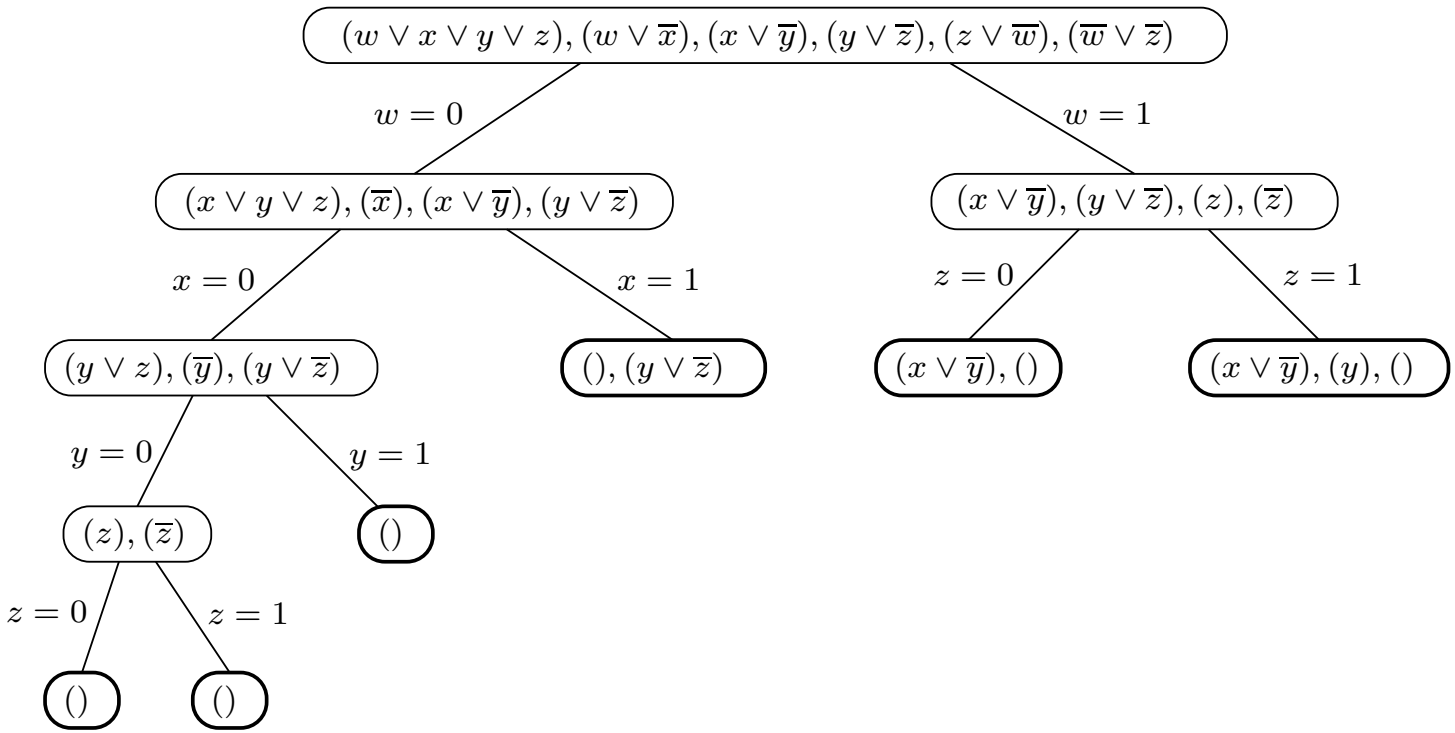


Figure: Backtrack search. Formula is not satisfiable.

Figure taken from book.

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

- pick variable that occurs in most clauses first
- pick variable that appears in most size 2 clauses first
- ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

- Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
- Run obvious test and in addition if all clauses are of size 2 then run 2-SAT polynomial time algorithm
- ...

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

- pick variable that occurs in most clauses first
- pick variable that appears in most size 2 clauses first
- ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

- Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
- Run obvious test and in addition if all clauses are of size 2 then run 2-SAT polynomial time algorithm
- ...

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

- pick variable that occurs in most clauses first
- pick variable that appears in most size 2 clauses first
- ...

What are quick tests for Satisfiability?

Depends on known special cases and heuristics. Examples.

- Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
- Run obvious test and in addition if all clauses are of size 2 then run 2-SAT polynomial time algorithm
- ...

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

- pick variable that occurs in most clauses first
- pick variable that appears in most size 2 clauses first
- ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

- Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
- Run obvious test and in addition if all clauses are of size **2** then run 2-SAT polynomial time algorithm
- ...

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be used also for optimization problems. Consider a minimization problem.

Notation: for instance **I**, **opt(I)** is optimum value on **I**.

$\mathbf{P_0}$ initial instance of given problem.

- Keep track of the best solution value **B** found so far. Initialize **B** to be crude upper bound on **opt(I)**.
- Let **P** be a subproblem at some stage of exploration.
- If **P** is a complete solution, update **B**.
- Else use a lower bounding heuristic to quickly/efficiently find a lower bound **b** on **opt(P)**.
  - If $\mathbf{b} \geq \mathbf{B}$ then prune **P**
  - Else explore **P** further by breaking it into subproblems and recurse on them.
- Output best solution found.

Example: Vertex Cover

Given $\mathbf{G = (V, E)}$, find a minimum sized vertex cover in $\mathbf{G}$.

- Initialize $\mathbf{B = n - 1}$.
- Pick a vertex $\mathbf{u}$. Branch on $\mathbf{u}$: either choose $\mathbf{u}$ or discard it.
- Let $\mathbf{b_1}$ be a lower bound on $\mathbf{G_1 = G - u}$.
- If $\mathbf{1 + b_1 < B}$, recursively explore $\mathbf{G_1}$
- Let $\mathbf{b_2}$ be a lower bound on $\mathbf{G_2 = G - u - N(u)}$ where $\mathbf{N(u)}$ is the set of neighbors of $\mathbf{u}$.
- If $\mathbf{|N(u)| + b_2 < B}$, recursively explore $\mathbf{G_2}$
- Output $\mathbf{B}$.

How do we compute a lower bound?
One possibility: solve an LP relaxation.

Example: Vertex Cover

Given $\mathbf{G = (V, E)}$, find a minimum sized vertex cover in $\mathbf{G}$.

- Initialize $\mathbf{B = n - 1}$.
- Pick a vertex $\mathbf{u}$. Branch on $\mathbf{u}$: either choose $\mathbf{u}$ or discard it.
- Let $\mathbf{b_1}$ be a lower bound on $\mathbf{G_1 = G - u}$.
- If $\mathbf{1 + b_1 < B}$, recursively explore $\mathbf{G_1}$
- Let $\mathbf{b_2}$ be a lower bound on $\mathbf{G_2 = G - u - N(u)}$ where $\mathbf{N(u)}$ is the set of neighbors of $\mathbf{u}$.
- If $\mathbf{|N(u)| + b_2 < B}$, recursively explore $\mathbf{G_2}$
- Output $\mathbf{B}$.

How do we compute a lower bound?
One possibility: solve an LP relaxation.

Local Search

Local Search: a simple and broadly applicable heuristic method

- Start with some arbitrary solution **s**
- Let **N(s)** be solutions in the "neighborhood" of **s** obtained from **s** via "local" moves/changes
- If there is a solution $\mathbf{s'} \in \mathbf{N(s)}$ that is better than **s**, move to $\mathbf{s'}$ and continue search with $\mathbf{s'}$
- Else, stop search and output **s**.

Local Search

Main ingredients in local search:

- Initial solution.
- Definition of neighborhood of a solution.
- Efficient algorithm to find a good solution in the neighborhood.

Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

2-change local search:

- Start with an arbitrary tour s_0
- For a solution s define s' to be a neighbor if s' can be obtained from s by replacing two edges in s with two other edges.
- For a solution s at most $O(n^2)$ neighbors and one can try all of them to find an improvement.

Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

2-change local search:

- Start with an arbitrary tour s_0
- For a solution s define s' to be a neighbor if s' can be obtained from s by replacing two edges in s with two other edges.
- For a solution s at most $O(n^2)$ neighbors and one can try all of them to find an improvement.

TSP: 2-change example

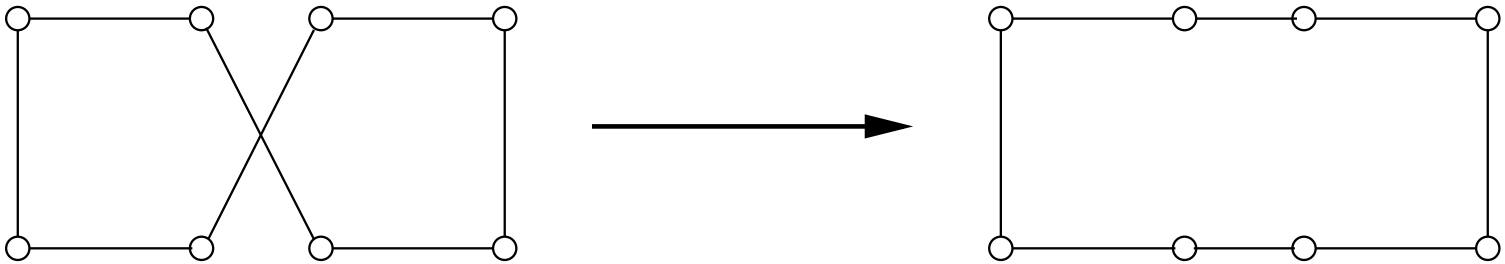


Figure below shows a bad local optimum for 2-change heuristic

TSP: 2-change example

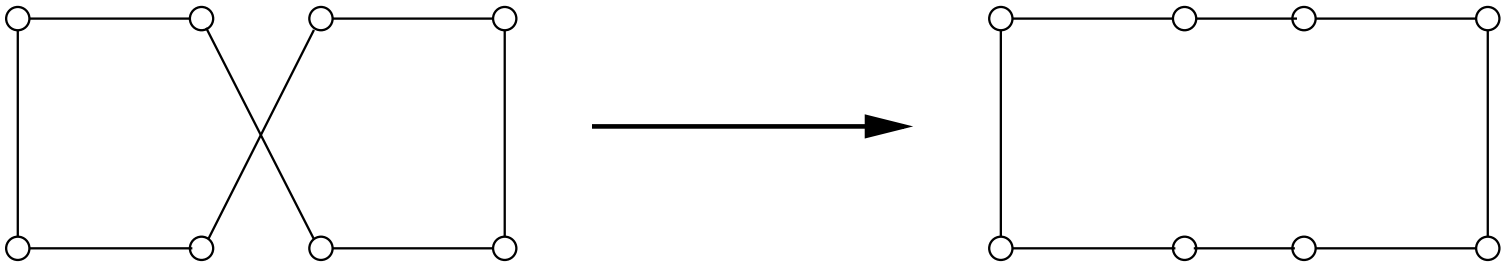
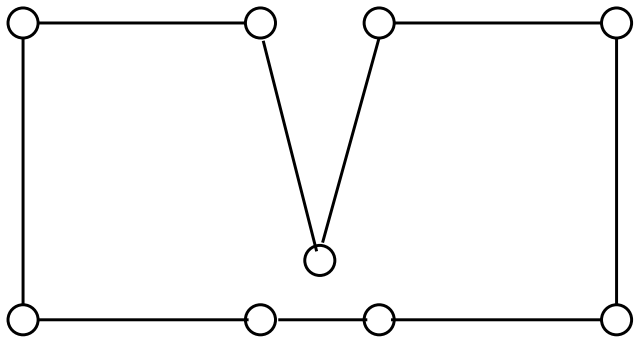
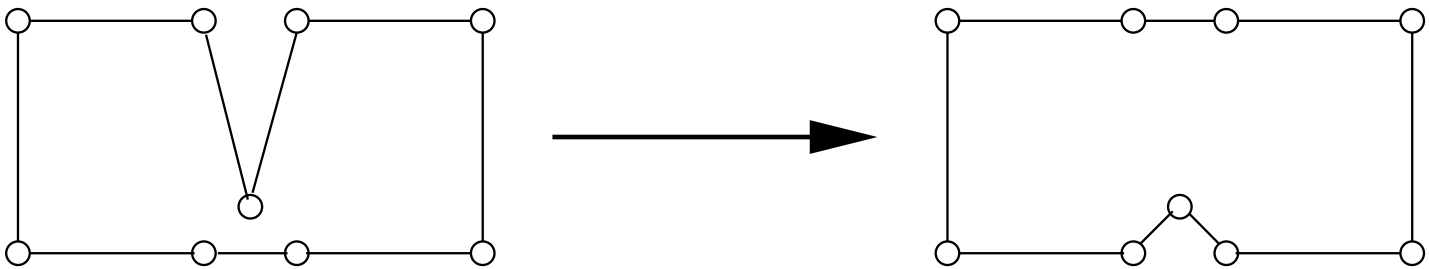


Figure below shows a bad local optimum for **2**-change heuristic



TSP: 3-change example

3-change local search: swap **3** edges out.



Neighborhood of **s** has now increased to a size of $\Omega(n^3)$
Can define **k**-change heuristic where **k** edges are swapped out.
Increases neighborhood size and makes each local improvement step less efficient.

Local Search Variants

Local search terminates with a local optimum which may be far from a global optimum. Many variants to improve plain local search.

- Randomization and restarts. Initial solution may strongly influence the quality of the final solution. Try many random initial solutions.
- Simulated annealing is a general method where one allows the algorithm to move to worse solutions with some probability. At the beginning this is done more aggressively and then slowly the algorithm converges to plain local search. Controlled by a parameter called "temperature".
- Tabu search. Store already visited solutions and do not visit them again (they are "taboo").

Heuristics

Several other heuristics used in practice.

- Heuristics for solving integer linear programs such as cutting planes, branch-and-cut etc are quite effective.
- Heuristics to solve SAT (SAT-solvers) have gained prominence in recent years
- Genetic algorithms
- ...

Heuristics design is somewhat ad hoc and depends heavily on the problem and the instances that are of interest. Rigorous analysis is sometimes possible.

Approximation algorithms

Consider the following *optimization* problems:

- **Max Knapsack**: Given knapsack of capacity **W**, **n** items each with a value and weight, pack the knapsack with the most profitable subset of items whose weight does not exceed the knapsack capacity.
- **Min Vertex Cover:** given a graph $G = (V, E)$ find the minimum cardinality vertex cover.
- **Min Set Cover:** given Set Cover instance, find the smallest number of sets that cover all elements in the universe.
- **Max Independent Set:** given graph $G = (V, E)$ find maximum independent set.
- **Min Traveling Salesman Tour:** given a directed graph **G** with edge costs, find minimum length/cost Hamiltonian cycle in **G**.

Solving one in polynomial time implies solving all the others.

Approximation algorithms

However, the problems behave very differently if one wants to solve them *approximately*.

Informal definition: An approximation algorithm for an optimization problem is an efficient (polynomial-time) algorithm that *guarantees* for every instance a solution of some given quality when compared to an optimal solution.

Some known approximation results

- **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a **0.99**-approximation efficiently.
- **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most **2** times the cost of an optimum solution.
- **Min Set Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most $(\ln n + 1)$ times the cost of an optimal solution.
- **Max Independent Set**: Unless **P = NP**, for any fixed $\epsilon > 0$, no polynomial time algorithm can give a $n^{1-\epsilon}$ relative approximation . Here n is number of vertices in the graph.
- **Min TSP**: No polynomial factor relative approximation possible.

Approximation algorithms

- Although **NP-Complete** problems are all equivalent with respect to polynomial-time solvability they behave quite differently under approximation (in both theory and practice).
- Approximation is a useful lens to examine **NP-Complete** problems more closely.
- Approximation also useful for problems that we can solve efficiently:
  - We may have other constraints such a space (streaming problems) or time (need linear time or less for very large problems)
  - Data may be uncertain (online and stochastic problems).

Part II

Closing Thoughts

Topics I wish I had time for

- More on data structures, especially use of amortized analysis
- Basic lower bounds on sorting and related problems
- Linear Programming
- More on heuristics and applications
- Experimental evaluation

Theoretical Computer Science

- Algorithms: find efficient ways to solve particular problems or broad category of problems
- Computational Complexity: understand nature of computation — classification of problems into classes (**P**, **NP**, **co-NP**) and their relationships, limits of computation.
- Logic, Languages and Formal Methods

Form the foundations for computer "science"

The Computational Lens

The Algorithm: Idiom of Modern Science by Bernard Chazelle
http://www.cs.princeton.edu/chazelle/pubs/algorithm.html

Computation has gained ground as *fundamental* artifact in mathematics and science.

- nature of proofs, **P** vs **NP**, complexity, ...
- quantum computation and information
- computational biology and the biological processes, ...

Standard question in math and sciences: Is there a *solution/algorithm*?
New: Is there an *efficient* solution/algorithm?

The Computational Lens

The Algorithm: Idiom of Modern Science by Bernard Chazelle
http://www.cs.princeton.edu/chazelle/pubs/algorithm.html

Computation has gained ground as *fundamental* artifact in mathematics and science.

- nature of proofs, **P** vs **NP**, complexity, ...
- quantum computation and information
- computational biology and the biological processes, ...

Standard question in math and sciences: Is there a *solution/algorithm*?
New: Is there an *efficient* solution/algorithm?

Related theory courses

- Graduate algorithms (currently being taught, every year)
- Computational complexity (next semester, every year)
- Randomized algorithms (currently being taught, every other year)
- Approximation algorithms (next semester, every other year)
- Advanced data structures (next semester, every once in a while)
- Cryptography and related topics (this semester, almost every year)
- Algorithmic game theory, combinatorial optimization, computational geometry, logic and formal methods, coding theory, information theory, graph theory, combinatorics, ...

And...

Questions?

Final Exam: ???, ??? ??th, 1.30 - 4.30pm in ???.

Thanks!

And...

Questions?

Final Exam: ???, ??? ??th, 1.30 - 4.30pm in ???.

Thanks!

Notes

Notes

Notes

Notes