15 – Complexity

PSPACE - A Complexity Class Beyond NP

Extending the Limits of Tractability

1. Computational resources

- Time:
  Complexity classes *P* and *NP*

- Space: Memory requirements of an algorithm
  An algorithm can reuse memory cells. Hence some complex problems can be solved using small space requirements.

Complexity classes

- *P*: An algorithmic problem belongs to the complexity class *P* of polynomially solvable problems if it can be solved by an algorithm with polynomial worst-case running time.

- *NP*: A decision problem Π belongs to the class *NP* (nondeterministic polynomial time) if there is a nondeterministic algorithm with polynomially bounded worst-case running time that accepts every $x \in \Pi$ along at least one computation path, and rejects every $x \notin \Pi$ along every computation path.

Polynomial running time: Polynomial in the size / encoding length of the input.

PSPACE

- *PSPACE*: An problem belongs to the class *PSPACE* if it can be solved by an algorithm with a polynomial worst-case amount of space.

Proposition: $P \subseteq PSPACE$

Proof: In polynomial time an algorithm can only consume a polynomial amount of space.

Example

Algorithm that counts from 0 to 2^n-1 in base-2 notation, using an n-bit counter.

Input length: n

Running time: 2^n

Space: n

Satisfiability

SAT: Boolean variables $x_1, \ldots, x_n$ with literals $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$. A clause $C = l_1 \vee \ldots \vee l_k$ is a disjunction of literals. Given a formula

$$\Phi = C_1 \wedge \ldots \wedge C_m,$$

does there exist an assignment of the Boolean variables (truth assignment) satisfying Φ?

3-SAT: Each clause has length three.

Example: $\Phi(x_1, x_2, x_3) =$

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Satisfiability

Proposition: 3-SAT $\in$ *PSPACE*.

Proof: Consider the brute-force algorithm that tries all the possible 2^n truth assignments.

Increment a binary n-bit counter from 0 to 2^n-1.

In each iteration the counter value $x_1, \ldots, x_n$ represents the values of the Boolean variables. Using polynomial space the algorithm can plug the values into the clauses and check if the formula is satisfied.

If so, the algorithm stops. Otherwise it erases the work space and moves on to the next truth assignment.

NP and PSPACE

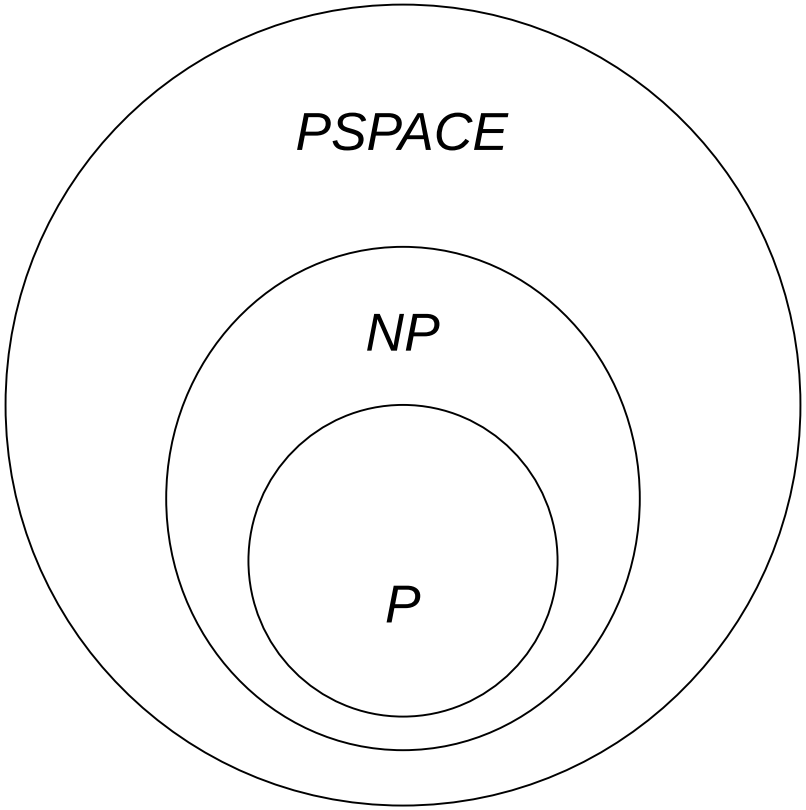
Polynomial reduction $\leq_p$: Decision problem *A* is polynomial time reducible to decision problem *B* (i.e. $A \leq_p B$) if there is a polynomial-time computable function *f* mapping instances of *A* to instance of *B* such that, for all *I*,

$$I \in A \Leftrightarrow f(I) \in B.$$

Theorem: *NP* $\subseteq$ *PSPACE*

Proof: 3-SAT is NP-complete and thus $\Pi \leq_p$ 3-SAT, for every problem $\Pi \in$ NP. Hence in polynomial time and space, every input *I* for Π can be transformed into an input *f*(*I*) for 3-SAT such that $I \in \Pi$ holds if and only if $f(I) \in$ 3-SAT. Input *f*(*I*) can be decided using a polynomial amount of space, by the above proposition.

Landscape



Hard problems in PSPACE

Quantification

QSAT (Quantified SAT): Boolean variables $x_1, \ldots, x_n$ with literals $\{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$. Does

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \; \Phi(x_1, \ldots, x_n)$$

hold? Here each Q_i is ∃ or ∀, for $1 \le i \le n$, and $\Phi(x_1, \ldots, x_n)$ is a 3-SAT formula.

Example: $\exists x_1 \forall x_2 \exists x_3 \; \Phi(x_1, x_2, x_3)$

where $\Phi(x_1, x_2, x_3) =$

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

$x_1 = 1 \;\; x_2 = 1 \;\; x_3 = 0$

$x_1 = 1 \;\; x_2 = 0 \;\; x_3 = 1$

Quantification

May assume that the quantifiers ∃ and ∀ alternate. Otherwise we add dummy variables.

$\ldots \exists x_i \, \exists x_{i+1} \ldots \Phi(x_1, \ldots, x_n)$ holds iff $\ldots \exists x_i \, \forall y_i \, \exists x_{i+1} \ldots \Phi(x_1, \ldots, x_n)$

$\ldots \forall x_i \, \forall x_{i+1} \ldots \Phi(x_1, \ldots, x_n)$ holds iff $\ldots \forall x_i \, \exists y_i \, \forall x_{i+1} \ldots \Phi(x_1, \ldots, x_n)$

An algorithm for QSAT

If the first quantifier is $\exists x_1$, then sequentially consider two cases.

- $x_1 = 0$: Recursively check whether the remaining quantified expression evaluates to 1.
- $x_1 = 1$: Recursively check whether the remaining quantified expression evaluates to 1.

Expression evaluates to 1 if one of the recursive calls evaluates to 1.

A quantifier $\forall x_1$ can be handled analogously but both recursive calls must evaluate to 1.

If we were to safe all the work in the recursive calls, then the space usage $S(n)$ would be $S(n) = 2 \cdot S(n\text{-}1) + p(n)$, for some polynomial function $p(n)$. This would result in an exponential bound.

Pseudocode

Algorithm QSAT

1. **if** next quantifier is $\exists x_i$ **then**
2. x_i := 0; recursively evaluate the quantified expression over the remaining variables; save the result (0 or 1) and delete all intermediate work;
3. x_i := 1; recursively evaluate the quantified expression over the remaining variables; save the result (0 or 1) and delete all intermediate work;
4. **if** either outcome yielded 1 **then** return 1 **else** return 0 **endif;**
5. **else if** next quantifier is $\forall x_i$ **then**
6. x_i := 0; recursively evaluate the quantified expression over the remaining variables; save the result (0 or 1) and delete all intermediate work;
7. x_i := 1; recursively evaluate the quantified expression over the remaining variables; save the result (0 or 1) and delete all intermediate work;
8. **if** both outcomes yielded 1 **then** return 1 **else** return 0 **endif;**
9. **endif;**

Analysis

$S(n)$ = space requirement on an n-variable problem

There holds

$$S(n) \leq S(n\text{-}1) + p(n),$$

for some polynomial function $p(n)$.

Hence $S(n) \leq p(n) + p(n\text{-}1) + p(n\text{-}2) + \ldots + p(1) \leq n \cdot p(n)$.

Theorem: QSAT $\in$ *PSPACE*.

PSPACE-completeness

Definition: A problem Π is *PSPACE-complete* if (a) Π is in *PSPACE* and (b) $\Pi' \leq_p \Pi$, for every problem Π' in *PSPACE*.

Theorem: QSAT is *PSPACE*-complete.

Proof: (Sketch) We need to show $\Pi \leq_p$ QSAT, for every $\Pi \in$ *PSPACE*.

The proof is similar to that of Cook's theorem. For Π, there exists a Turing machine *M* that decides Π using a polynomial amount of space.

Suppose that an input $I \in \{0,1\}^n$ can be decided using space $p(n)$, for some polynomial function p. In polynomial time one can construct a quantified Boolean formula of size $O(p(n)^2)$ that is true if and only if *M* accepts *I*.

Games

A large number of two-player games, such as chess, naturally fit into the following framework. Players alternate moves, and the first one to achieve a specfic goal wins.

Problem Competitive Facility Location: $G=(V,E)$ undirected graph. Vertex v_i has a non-negative real value b_i. Non-negative bound B. Two players alternately select vertices so that the selected vertices always form an independent set.

Player 2 wins if he can select a set of vertices of total value at least B. Otherwise Player 1 wins.

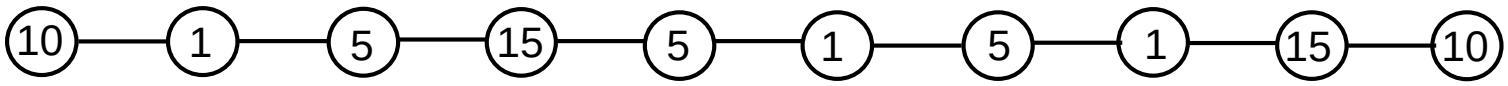
Can Player 1 (or Player 2) force a win?

Application

Two companies, JavaPlanet and Queen's Coffee, operating café franchises across the country, compete for market share in a geographic region.

The region is divided into n zones labeled $1,2,\ldots,n$. Each zone i has a value b_i, representing the revenue if a company opens a franchise there. Local zoning laws require that a pair (i,j) of adjacent zones must not each contain a franchise. This can be modeled by a conflict graph $G=(V,E)$.

The two companies/players take turn in selecting zones. Can Player 2 select zones of total value at least B?



B= 20? B= 25?

Analysis

Theorem: Competitive Facility Location $\in$ *PSPACE*.

Proof: Similar to the proof that QSAT $\in$ *PSPACE*.

Consider the alternate moves of the two players.

Player 1: Check all possible moves. For each one, determine if it results in a forced win for Player 1 in the resulting game. Player 1 has a forced win in the current position if each of the moves yields a forced win.

Player 2: Check all possible moves. For each one, determine if it results in a forced win for Player 2 in the resulting game. Player 2 has a forced win in the current position if at least one of the moves yields a forced win.

In each case, a single bit suffices to store the result.

Analysis

$S(n)$ = space requirement on a graph with n vertices

There holds

$$S(n) \leq S(n\text{-}1) + p(n),$$

for some polynomial function $p(n)$.

Planning

Fundamental problem in Artificial Intelligence.

Arises in disaster-relief efforts, military operations and solitaire puzzles, such as Rubic‘s Cube or the Fifteen Puzzle.

Given an initial state, is it possible to apply a sequence of operations so as to reach a desired final state.

12	15	6	10
4	9	5	8
14	13		2
1	7	11	3

Definition of the Planning problem

- Conditions $C = \{c_1,\ldots,c_n\}$

  A configuration is specified by a subset of the conditions.
- Operators $O = \{o_1,\ldots,o_k\}$

  Each operator o_i has

  prerequisite list p_i: conditions that must hold for o_i to be invoked

  add list a_i: conditions that become true after o_i is invoked

  delete list d_i: conditions that cease to hold after o_i is invoked.

Given a set C_0 of initial conditions and a set of goal conditions C^*, is it possible to apply a sequence of operators so as to reach C^* from C_0?

Graph representation

$G=(V,E)$

V: Vertex for each of the 2^n possible configurations

E: There is a directed edge from C' to C'' if in one step one of the operators converts C' to C'', for any two configurations C' and C''.

Does there exists a path from C_0 to C^*?

Worst-case instances

Lemma: There are instances of the Planning Problem with n conditions and k operators for which there exists a solutions but the shortest one has length 2^n-1.

Proof:

- Conditions $c_1,\ldots,c_n$.
- Operators $o_1,\ldots,o_n$.
- o_1 has no prerequisite or delete list. It adds c_1.
- o_i, where $i>1$, requires c_j for all $j<i$ as prerequisite.
  It adds c_i and deletes all c_j, for all $j<i$.

$C_0 = \emptyset$ $\quad C^* = \{c_1,\ldots,c_n\}$

Worst-case instances

Claim: For any configuration that does not contain c_j, for all $j \leq i$, there exists a sequence of operators that reaches a configuration containing c_j, for all $j \leq i$, but any such sequence has at least 2^i-1 steps.

Proof: By induction on i. The claim holds for i=1. Consider i>1.

Existence of a solution:

- Achieve $\{c_1,\ldots,c_{i-1}\}$, using $o_1,\ldots,o_{i-1}$.
- Invoke o_i, adding c_i but deleting everything else.
- Achieve again $\{c_1,\ldots,c_{i-1}\}$, using $o_1,\ldots,o_{i-1}$. This preserves c_i.

Worst-case instances

Lower bound on the number of steps.

Consider the first moment c_i is added. At that time $c_1,\ldots,c_{i-1}$ must be present. This requires at least 2^{i-1}-1 steps. Operator o_i is invoked, which takes one step and erases $c_1,\ldots c_{i-1}$. At least 2^{i-1}-1 further steps are needed to restore $c_1,\ldots,c_{i-1}$.

The lemma follows by applying the claim for $i=n$.

Worst-case instances

By the above lemma, depth-first and breadth-first search cannot be used to find a solution using a polynomial amount of space.

Lemma: If a Planning instance with n conditions has a solution, then it has one using at most 2^n-1 steps.

Space-efficient path construction

We present a path finding algorithm, based on an idea proposed Savitch in1970.

Procedure Path(C_1,C_2,L): It determines whether there is a sequence of operators, consisting of at most L steps, that leads from configuration C_1 to configuration C_2.
Generate all possible midpoints C‘ and check recursively whether one can get from C_1 to C‘ in $L/2$ steps and from C‘ to C_2 in $L/2$ steps.

This involves two recursive calls. Only the outcome yes/no matters so that we can reuse space.

Pseudocode

Algorithm Path(C_1,C_2,L)

1. **if** $L = 1$ **then**
2. **if** there is an operator converting C_1 to C_2 **then**
3. return „yes“;
4. **else** return „no“;
5. **endif;**
6. **else**
7. Enumerate all configurations C‘ using an n-bit counter;
8. **for** each C‘ **do**
9. x := Path(C_1,C‘,$\lceil L/2 \rceil$); Delete all intermediate work, saving only x;
10. y := Path(C‘,C_2,$\lfloor L/2 \rfloor$); Delete all intermediate work, saving only y;
11. **if** x and y are both „yes“ **then** return „yes“; **endif;**
12. **endfor;**
13. **if** „yes“ was not returned for any C‘ **then** return „no“; **endif;**
14. **endif;**

Space-efficient path construction

Lemma: Path(C_1,C_2,L) returns „yes“ if and only if there is a sequence of operators of length at most L leading from C_1 to C_2 . The space requirement is polynomial in n, k and log L.

Proof: The correctness follows by induction on L.

L=1: All operators can be checked explicitly.

L>1: If there is a sequence of operators from C_1 to C_2 of length $L' \leq L$, then there exists a configuration C' that occurs at position $\lceil L'/2 \rceil$ in this sequence. By induction Path(C_1,C',$\lceil L/2 \rceil$) and Path(C',C_2,$\lfloor L/2 \rfloor$) both return „yes“ so that Path(C_1,C_2,L) also returns „yes“.

Conversely, if Path(C_1,C_2,L) returns „yes“, then there exists a configuration C' so that Path(C_1,C', $\lceil L/2 \rceil$) and Path(C',C_2,$\lfloor L/2 \rfloor$) both return „yes“. By induction hypothesis the exist corresponding sequences from C_1 to C' and from C' *to* C_2. Their concatenation gives a sequence from C_1 to C_2 of length at most L.

Space-efficient path construction

We next analyze the space requirements. In addition to the space needed inside a recursive call, each invocation of the procedure Path uses an amount of space that is polynomial in n, k and log L.

Hence the space requirements $S(n, k, L)$ satisfies

$S(n, k, 1) \leq p(n, k, 1)$

$S(n, k, L) \leq S(n, k, \lceil L/2 \rceil) + p(n, k, \log L)$

for some polynomial function p.

This yields $S(n, k, L) = O(\log L \cdot p(n, k, \log L))$.

Theorem: Planning $\in$ *PSPACE*.

Competitive Facility Location

Problem: $G=(V,E)$ undirected graph. Vertex v_i has a non-negative real value b_i. Non-negative bound B. Two players alternately select vertices so that the selected vertices always form an independent set.

Player 2 wins if he can select a set of vertices of total value at least B. Otherwise Player 1 wins.

Can Player 1 force a win?

Proving problems PSPACE-complete

Theorem: Competitive Facility Location is *PSPACE*-complete.

Proof: We will show QSAT $\leq_p$ Competitive Facility Location.

We are given an instance $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$ of QSAT. For simplicity we assume that n is odd. The formula is a conjunction of clauses $C_1 \wedge \ldots \wedge C_k$, where each C_j can be written as

$$C_j = l_{j1} \vee l_{j2} \vee l_{j3}.$$

Moreover, we assume that no clause contains a variable and its negation.

Given $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$, we define a graph G for Competitive Facility Location. In QSAT the variables $x_1, \ldots, x_n$ are set in this order. Competitive Facility Location is more general in that players may select arbitrary vertices as long as they form an independent set. We will fix this by assigning specific values to the vertices.

Proving problems PSPACE-complete

Graph construction: For each variable x_i, we introduce two vertices v_i and v_i', representing literals x_i and $\bar{x}_i$, respectively. The two vertices are connected by an edge $\{v_i, v_i'\}$.

Selecting v_i corresponds to $x_i = 1$. Selecting v_i' corresponds to $x_i = 0$.

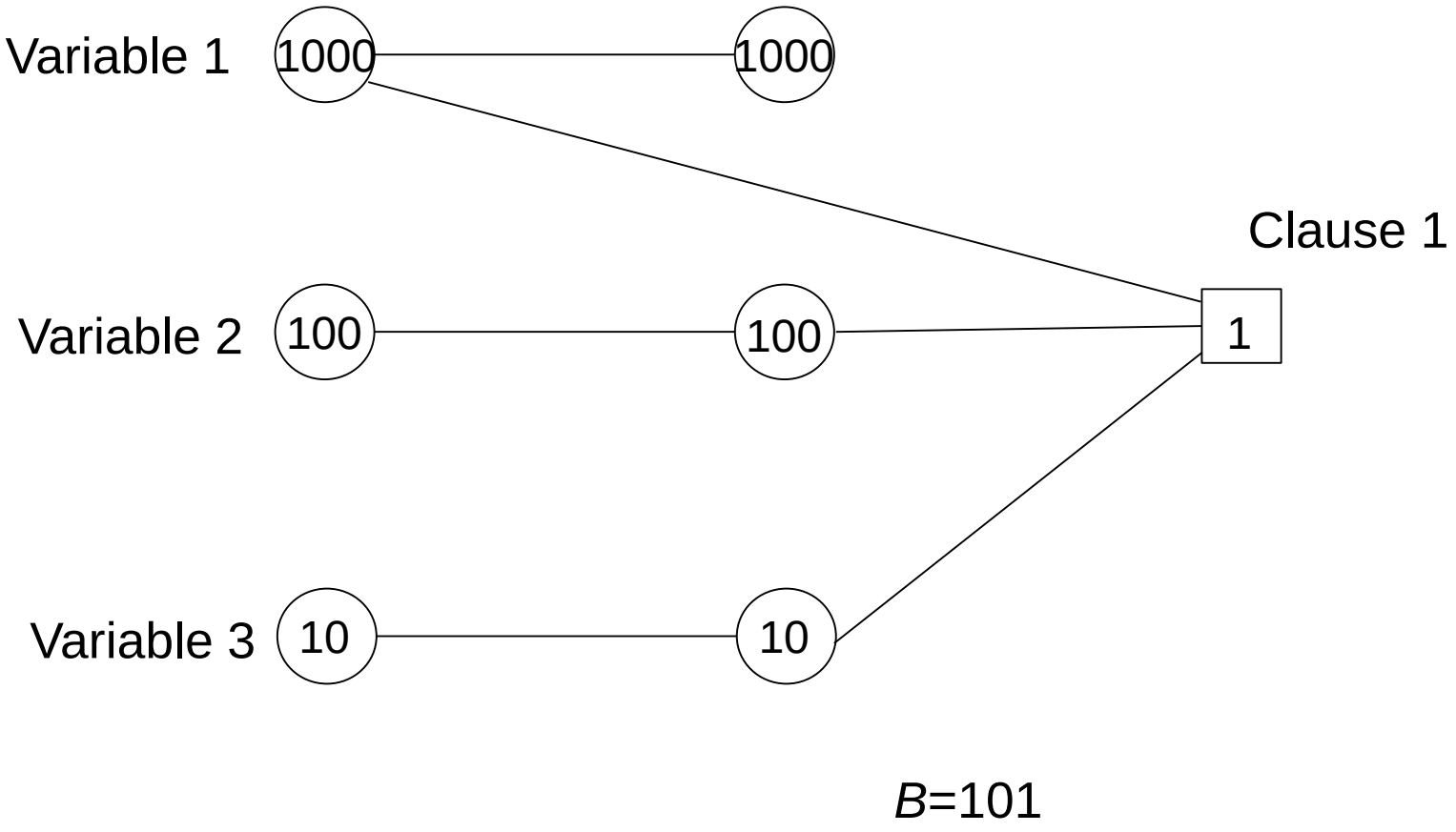
In order to ensure that vertices are picked in increasing order of index, a high value is assigned to v_1 and v_1'. Player 1 will lose instantly if he does not select any of the two vertices. A smaller value is assigned to v_2 and v_2', and so on.

Let $c > 2$ be a constant. The b-values of v_i and v_i' are equal to c^{1+n-i}. The bound to be achieved by Player 2 is

$$B = c^{n-1} + c^{n-3} + \ldots + c^2 + 1.$$

For each clause C_j we introduce a vertex w_j that is connected to those literals contained in that clause and has value 1.

Proving problems PSPACE-complete



Proving problems PSPACE-complete

Suppose that during the first *i*-1 rounds, the players alternately chose vertices in increasing order of index. Assume that in round *i*, the corresponding player does not select a vertex from among v_i and v_i'.

i odd: Player 2 wins instantly in the next round by picking either v_i or v_i' because $c^{1+n-i} > c^{n-i} + \ldots + c^2 + 1$.

i even: Player 1 wins instantly in the next round by picking either v_i or v_i' because the remaining value that can be collected by Player 2 is at most $c^{n-i} + \ldots + c^2 + 1 < c^{1+n-i}$.

We prove that $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$ holds if and only if Player 1 has a forced win.

Proving problems PSPACE-complete

First assume that $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$ holds. Consider any of the rounds *i* where Player 1 moves. If in the previous round, for the first time, Player 2 has not selected either v_{i-1} or v'_{i-1}, then Player 1 wins instantly.

Otherwise Player 1 inspects $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$ with the prior setting of $x_1, \ldots, x_{i-1}$; the choice of v_j, for any $j < i$, corresponds to x_j=1. Player 1 selects v_i if $x_i = 1$ and v'_i otherwise.

In each clause of $\Phi(x_1, \ldots, x_n)$ at least one literal gives true and a corresponding vertex has been selected. Hence after *n* rounds Player 2 cannot select any of the clause vertices w_j and its total value is $c^{n-1} + c^{n-3} + \ldots + c^2 = B-1$. Thus Player 1 has a forced win.

Proving problems PSPACE-complete

Next assume that $\exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n)$ does not hold. Then $\forall x_1 \exists x_2 \ldots \exists x_{n-1} \forall x_n$ formula $\Phi(x_1, \ldots, x_n)$ is not satisfied.

Now consider any of the rounds *i* where Player 2 moves. As above, Player 2 wins instantly or he selects v_i if $x_i = 1$ and v_i' otherwise. In $\Phi(x_1, \ldots, x_n)$ at least one clause C_j is not satisfied and Player 2 can select that clause vertex w_j, earning a total value of *B*.

Hence Player 2 has a forced with, and Player 1 cannot force a win.

Extending the limits of tractability

- Develop algorithms for NP-complete decision problems / NP-hard optimization problems with a reasonable efficiency.

- Fixed-Parameter Algorithms: Running time is exponential in a fixed problem parameter but polynomial otherwise.

  The hope is that, for typical input instances, the chosen parameter takes small values. The algorithms exploit special structures of the given problem.

Vertex Cover

Problem Vertex Cover: Graph $G=(V,E)$ and integer k.
A vertex cover is a subset $S \subseteq V$ of the vertices such that every edge $e \in E$ has at least one endpoint in S. Is there a vertex cover of size $|S| \leq k$?

Two problem parameters: $n = |V|$ and k.

For constant k, Vertex Cover can be solved in polynomial time: Check all the $\binom{n}{k}$ subsets of size k. For each one it takes $O(kn)$ time to find out if it is a vertex cover. The total running time is $O(kn\binom{n}{k}) = O(kn^{k+1})$.

This is impractical. For example, if n=1000 and k=10, a computer executing 10^6 instructions per second takes 10^{25} seconds. This is larger than the age of the universe.

Will develop an algorithm having a running time of $O(2^k kn)$. For n=1000 and k=10, the time is still bounded by a few seconds.

Vertex Cover

Lemma: If in G the maximum degree of any vertex is at most d and there is a vertex cover of size at most k, then G has at most kd edges.

Proof: Let S be a vertex cover in G of size $k' \leq k$. Every edge in G has at least one endpoint in S. However, each vertex in S can cover at most d edges. Thus there are at most $k'd \leq kd$ edges in G.

Corollary: If G has a vertex cover of size k, then it has at most $k(n-1) \leq kn$ edges.

Algorithm for Vertex Cover

In a first step the algorithm checks if G has more than kn edges. If so, the graph does not have a vertex cover of size k. Otherwise the algorithm proceeds.

For any $v \in V$, let $G\backslash\{v\}$ be the graph obtained from G by deleting v and its incident edges.

Lemma: Let $\{u,v\}$ be any edge of G. Graph G has a vertex cover of size at most k if and only if at least one of the graphs $G\backslash\{u\}$ and $G\backslash\{v\}$ has a vertex cover of size at most k-1.

Algorithm for Vertex Cover

Proof: Suppose that G has a vertex cover S of size at most k. For any edge $\{u,v\}$ in G, at least one of u and v is contained in S. Assume that it contains u. Then $S\setminus\{u\}$ must cover all edges that have neither endpoint equal to u. Hence $S\setminus\{u\}$ is a vertex cover of size k-1 for $G\setminus\{u\}$.

Conversely, suppose that one of $G\setminus\{u\}$ and $G\setminus\{v\}$ has a vertex cover of size at most k-1. Assume that $G\setminus\{u\}$ has such a cover T. Then $T \cup \{u\}$ covers all the edges of G.

Pseudocode

Algorithm Vertex Cover

Search for a vertex cover of size k in $G=(V,E)$.

1. **if** $|E| > k\,|V|$ **then** return "G has no vertex cover of size k" **endif**;
2. **if** $|E| = 0$ **then** return $\emptyset$ **endif**;
3. Let $\{u,v\}$ be any edge of G;
4. Recursively check if $G\setminus\{u\}$ or $G\setminus\{v\}$ has a vertex cover of size k-1;
5. **if** at least one of them, say $G\setminus\{u\}$, has such a cover T **then**
6. **return** $T \cup \{u\}$;
7. **else** return „G has no vertex cover of size k";
8. **endif;**

Analysis

Theorem: The running time of the Vertex Cover Algorithm is $O(2^k kn)$.

Proof:

$T(n,k)$ = running time on a graph with n vertices and parameter k

There holds $T(n,1) \leq cn$

$$T(n,k) \leq 2T(n,k\text{-}1) + cn,$$

for some constant c.

By induction on k we can show that $T(n,k) \leq c \cdot 2^k kn$ holds for all n.

If k=1, the inequality is satisfied for all n. Suppose that it holds for k-1.

Then $T(n,k) \leq 2T(n\text{-}1,k\text{-}1) + cn$

$$\leq 2c \cdot 2^{k-1}(k\text{-}1)n + cn$$

$$= c \cdot 2^k kn - c \cdot 2^k n + cn$$

$$\leq c \cdot 2^k kn.$$

Solving NP-hard problems on trees

Study NP-complete / NP-hard problems on inputs that are structurally simple.

Many NP-complete graph problems can be solved efficiently on trees.

Intuition: A subtree rooted at v forms an almost independent subproblem that interacts with the rest of the tree only through v.

Problem Independent Set: Given graph $G=(V,E)$, a set $S \subseteq V$ is independent if no two vertices of V are connected by an edge. Find a maximum-size independent set.

Independent Set on trees

Greedy approach: Every tree has a leaf, i.e. a vertex of degree 1.

Let v be a leaf and $\{u,v\}$ be the unique edge incident to v. An independent set contains either u or v.

Lemma: If $T = (V,E)$ is a tree and v is a leaf of T, then there exists a maximum-size independent set that contains v.

Proof: Consider a maximum-size independent set S and let $\{u,v\}$ be the unique edge incident to v. Set S contains one of u or v. If it contains v, we are done. Otherwise we can delete u and add v, thereby obtaining another independent set of the same size.

Algorithm

Greedy approach: The algorithm repeatedly identifies and deletes vertices that can be placed in the independent set.

The deletions may disconnect the tree. Hence the algorithm actually works on a forest, i.e. a graph in which each connected component is a tree.

We will describe the algorithm for a forest. An optimal solution for a forest is the union of the optimal solutions for each tree component.

Algorithm

Algorithm Independent Set

Find a maximum-size independent set in a forest F.

1. $S := \emptyset$;
2. **while** F has as least one edge **do**
3. Let $\{u,v\}$ be any edge of F such that v is a leaf;
4. $S := S \cup \{v\}$;
5. Delete from F vertices u, v and all edges incident to them;
6. **endwhile;**
7. **return** S;

Theorem: The above algorithm finds a maximum size independent set in a forest.

The algorithm can be implemented to run in linear time. One simply maintains at any time a list of the leaves.

Maximum-Weight Independent Set on trees

Weighted Independent Set in Trees: Tree $T=(V,E)$, where each vertex $v \in V$ has a positive real weight w_v. Find an independent set S so that the total weight $\sum_{v \in S} w_v$ is as large as possible.

Again consider an edge $\{u,v\}$, where v is a leaf.

Including v into the independent set blocks fewer vertices. However, if $w_v < w_u$, this might not be the best choice.

Both options, i.e. the inclusion of u or v, have to be considered. If u is not included in the independent set, then all of its children may be added.

Maximum-Weight Independent Set on trees

Dynamic programming approach: Root the tree at an arbitrary vertex *r*. Start at the leaves and gradually work up the tree.

For any vertex *u*,

solve the subproblem associated with the tree T_u rooted at *u*,

after the subproblems have been solved for all the children of *u*.

$OPT_{in}(u)$ = maximum weight of an independent set of T_u that includes *u*

$OPT_{out}(u)$ = maximum weight of an independent set of T_u that does not include *u*

$$OPT_{in}(u) = w_u + \sum_{v \in children(v)} OPT_{out}(v)$$

$$OPT_{out}(u) = \sum_{v \in children(v)} \max\{ OPT_{out}(v), OPT_{in}(v) \}$$

Pseudocode

Algorithm Maximum-Weight Independent Set

Find a maximum weight-independent set of a tree T.

1. Root the tree at a vertex r;
2. **for all** u of T in postorder **do**
3. **if** u is leaf **then**
4. $M_{out}[u] := 0$; $M_{in}[u] := w_u$;
5. **else**
6. $M_{out}[u] := w_u + \sum_{v \in children(v)} M_{out}[v]$;
7. $M_{in}[u] := \sum_{v \in children(v)} \max\{ M_{out}[v], M_{in}[v]\}$;
8. **endif;**
9. **return** $\max\{ M_{out}[r], M_{in}[r]\}$;

Arrays $M_{out}[u]$ and $M_{in}[u]$ hold the values $OPT_{in}(u)$ and $OPT_{in}(u)$, respectively.

We can recover the maximum-weight independent set by recording, for each vertex, the decision taken.

Result

Theorem: The above algorithm finds a maximum-weight independent set in trees in linear time.