CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 23: Complexity of Games and Puzzles

Harvard SEAS - Fall 2023 *2023-11-28*

1 Announcements

- PS9 out later today
- PS8 due tomorrow.
- The material below is mostly for fun/culture. You do not need to be able to know these results at a technical level, although some of them are examples of things (e.g. proofs of NP-hardness) that you should be familiar with.
- Please fill out Q survey

2 Loose end from last time: unsolvability of 3Coloring

Our next example of an unsolvable problem is the following:

Input : A RAM program P
Output : `yes` if P correctly solves the graph 3-coloring problem, `no` otherwise

Computational Problem Solves3Coloring

Theorem 2.1. *Solves3Coloring is unsolvable.*

Proof.
It suffices to show that HaltOnEmpty reduces to Solves3Coloring, i.e. there is an algorithm A that solves HaltOnEmpty given an oracle for Solves3Coloring. We follow the same reduction template as before:

1 $A(P)$:
  Input : A RAM program P
  Output : `yes` if P halts on ε, `no` otherwise
2 Construct from P a RAM program Q_P such that Q_P solves 3-coloring if and only if P halts on ε;
3 Run the Solves3Coloring oracle on Q_P and return its result;

Algorithm 1: Template for reduction from HaltOnEmpty to Solves3Coloring

This time, we construct the program Q_P as follows:

1 $Q_P(G)$:
  Input : A graph G
2 Run P on ε;
3 **return** *ExhaustiveSearch3Coloring(G)*

Algorithm 2: The RAM Program Q_P constructed from P

Similarly to our previous reduction, we need to check:

Claim 2.2. *Q_P solves 3-coloring if and only if P halts on ε.*

To verify the claim, note that if P doesn't halt on ε, the $Q_P(G)$ will never halt, and thus cannot solve 3-coloring. On the other hand, if P does halt on ε, then $Q_P(G)$ will always have the same output as ExhaustiveSearch3Coloring(G) and thus correctly solves 3-coloring.

Claim 2.2 implies that plugging this construction of Q_P in to Algorithm 3 gives a correct reduction from HaltsOnEmpty to Solves3Coloring, and thus completes the proof that Solves3Coloring is unsolvable.

□

3 Jigsaw puzzle: modeling

Definition 3.1. A *jigsaw puzzle* consists of natural numbers m and n, called its *dimensions*, and a set $S = \{t_0, t_1, \ldots, t_{mn-1}\}$ of "jigsaw tiles".

A *jigsaw tile* consists of four integers t.left, t.right, t.top, and t.bottom.

A *jigsaw partial solution* is a function $f : S \to ([m] \times [n]) \cup \{\text{"unassigned"}\}$ such that:

1. For all $t, t' \in S$, $f(t) \neq f(t')$ or $f(t) =$ "unassigned".
2. For all $t, t' \in S$, if $f(t) = f(t') + (0, 1)$, then f(t').top = -f(t).bottom.
3. For all $t, t' \in S$, if $f(t) = f(t') + (1, 0)$, then f(t').right = -f(t).left.

A *jigsaw solution* is a jigsaw partial solution in which no piece is "unassigned".

Question: What aspects of real-life jigsaw puzzles have we not modeled with the definitions above?

1. There may be multiple types of symmetric edges.
2. The edge of the board must be flat.
3. The puzzle has an image.
4. Edges that match closely enough might still fit.
5. Jigsaws might have solutions that aren't rectangular grids.
6. Tiles can be rotated.
7. ⋮

4 One-player jigsaws

We can state the problem of finding a solution to a jigsaw puzzle as a computational problem:

Input	**:** A jigsaw puzzle, as defined above.
Output	**:** A jigsaw solution, as defined above.

Computational Problem JigsawA1

One can think of solving a jigsaw puzzle as playing a 1-player game where the moves are placing jigsaw pieces. The "1" in the name of the computational problem reflects the one-player game.

Theorem 4.1. *JigsawA1 is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Proof. We need to check that JigsawA1 is in $\mathsf{NP}_{\mathsf{search}}$ and that it's $\mathsf{NP}_{\mathsf{search}}$-hard.

JigsawA1 is in $\mathsf{NP}_{\mathsf{search}}$: given a proposed jigsaw solution, we can verify that it's correct by checking the compatibility of each pair of adjacent pieces: that is, check the condition defining a jigsaw partial solution.

JigsawA1 is $\mathsf{NP}_{\mathsf{search}}$-hard. We'll prove so by reduction from LongPath, which you proved was $\mathsf{NP}_{\mathsf{search}}$-hard in SRE4. Recall the definition of LongPath:

Input	**:** A digraph $G = (V, E)$, two vertices $s, t \in V$, and a path-length $k \in \mathbb{N}$
Output	**:** A path from s to t in G of length k, if one exists

Computational Problem LongPath

In fact, LongPath is $\mathsf{NP}_{\mathsf{search}}$-hard even if $k = n - 1$, so we'll reduce from that problem. We'll also assume that no edges leave t, since those edges can't be used in any path ending at t.

Given an input to LongPath, we'll construct a jigsaw puzzle whose dimensions are $(n + 2m + 4) \times (n + 2m + 4)$. In particular, we'll choose tiles that make two gadgets: $(n + 2m + 4)^2 - m$ tiles which make a framework gadget, and m tiles which make a path-and-disposal gadget.

Framework gadget: Our goal for the framework gadget is to fill in all the space of the jigsaw puzzle except the following reserved spaces:

1. An $(n - 1) \times 1$ rectangle whose left edge is $-s$, whose right edge is t, all of whose top edges are -1, and all of whose bottom edges are -2.

2. $m - n + 1$ unit squares whose top edges are -1, bottom edges are -2, and left edges are the left edge of the puzzle (and are hence unconstrained). For each vertex $v \in V \setminus \{t\}$, if the number of edges leaving v is $\deg^+(v)$, then $\deg^+(v) - 1$ of the right edges of those squares should be -v.

Therefore, we create $(n + 2m + 4)^2 - m$ tiles, one tile $t_{x,y}$ for each pair $(x, y) \in [n + 2m + 4]^2$ other than the m reserved above. For each pair of these tiles $t_{x,y}$ and $t_{x,y+1}$, we set $t_{x,y}$.top to be some integer which doesn't appear as any other edge, and set $t_{x,y+1}$.bottom to its negative. Do similarly for each pair of these tiles $t_{x,y}$ and $t_{x+1,y}$. For each tile edge which should be on the edge of the board, we choose some integer for which neither it nor its negative appears on any other edge. For each tile edge which should border one of the reserved spaces, we set the corresponding side of the tile as defined in the reserved space.

Path-and-disposal gadget: For each edge $(u, v) \in E$, we make a tile t with t.top = 1, t.bottom = 2, t.left = u, and t.right = -v.

Proof of correctness: In the framework gadget, there are $m+2n+4$ numbers which appear as tops whose negatives don't appear in the puzzle, so the $m+2n+4$ tiles with those tops must be along the top edge. Each of the bottoms of those tiles only matches one other tile, which forces the placement of those tiles, and so on. All tiles in the framework gadget are connected to the top of the puzzle via tiles of the framework gadget, so in any solution, the tiles of the framework gadget must be placed in their intended places.

Each of the $m-n+1$ unit squares whose top edges are -1, bottom edges are -2, and left edges are the left edge of the puzzle (and are hence unconstrained), and right edge is $-v$ must be filled with a piece whose left edge is v. There are $\deg^+(v)$ pieces whose left edge is v (one for each edge leaving v in G), so any solution to the jigsaw must, after filling these, leave only one tile whose left edge is v, for each $v \in E \setminus \{t\}$.

If G has a path $(v_0, e_1, v_1, \ldots, e_{n-1}, v_{n-1})$ of length $n-1$, then we can form an $(n-1) \times 1$ rectangle out of the tiles corresponding to e_1, ..., e_{n-1} whose left edge is s, whose right edge is $-t$, all of whose top edges are 1, and all of whose bottom edges are 2, using one tile whose left edge is v, for each $v \in E \setminus \{t\}$.

Conversely, if we can form an $(n-1)\times 1$ rectangle with that set of edges out of some of the edge tiles, then the sequence of vertices on their vertical edges is a walk of length n from s to t: each pair of adjacent vertical edges are on a tile because there's an edge between them, so consecutive vertices of the walk are adjacent. Since this walk uses only one tile whose left edge is any vertex v, its vertices are all distinct, so it's a path of length $n-1$.

Therefore, the jigsaw puzzle is solvable if and only if G had a long path, and if G has a long path, we can solve the jigsaw puzzle by the piece placements described above.

□

5 Two-player jigsaws

We could define a two-player game version of jigsaw-solving: the players take turns placing a tile (starting with t_0, then t_1, etc.) adjacent to some already-assigned tile. Player 0 wants to solve the jigsaw puzzle and player 1 wants to prevent it from being solved.

Input	**:** A jigsaw puzzle
Output	**:** True if player 0 has a strategy in the two-player game version of jigsaw-solving which guarantees a win (i.e. a solved puzzle); false otherwise.

Computational Problem JigsawA2

JigsawA2 is complete for PSPACE, a complexity class we haven't defined yet:

Definition 5.1. PSPACE is the set of decision problems which can be solved in polynomial space. That is, PSPACE is the set of decision problems solved by some Word-RAM algorithm which, on inputs of size n, calls MALLOC at most $O(n^c)$ times, for some constant c.

Proving that JigsawA2 is PSPACE-hard is out of scope for CS 120, but we can prove that JigsawA2 is *in* PSPACE:

Theorem 5.2. *JigsawA2 is in PSPACE.*

Proof. A complete two-player jigsaw game lasts (at most) mn turns. For each $i \in [mn]$, we'll store in the ith cell $M[i]$ of memory, a move (i.e. a valid piece placement) that could be made in the ith turn.

Algorithm: We'll run mn nested loops: on the outside, we'll consider each possible move 0. For each of them, we'll consider each possible move 1 (by player 1). For each of those, we'll consider each possible move 2 (by player 0 again). In the innermost loop, we consider each possible move mn—there will be at most one legal move left at this point, since there's only one piece left to place. We'll have the innermost loop return True if the result is a jigsaw solution, and False otherwise.

If we're in iteration i_0 of the 0th loop, i_1 of the 1st loop, ..., and i_k of the kth loop, the $(k+1)$st loop returns some set X of Trues and some Falses. We return True from the i_kth iteration of the kth loop if either:

1. k is even and all values in X are True.
2. k is odd and at least one value in X is True.

Finally, if the 0th loop returns at least one True, our algorithm returns True; otherwise we return False.

Proof of Correctness: If the algorithm returns True, then a winning strategy for player 0 is to choose, at move k, some move for which the kth loop returned True. Such a move is available at move 0 because the algorithm returned True. If such a move is available at move k (where k is even, because Player 0 makes the even moves), then the set X of return values from the kth loop are True. That is, for any $(k+1)$st move which player 1 makes, the $(k+1)$st loop returns True for that move, which means that at least one $(k+2)$nd move is available for player 0 for which the loop returned True. At the end of the game, "returning True" corresponds to the jigsaw puzzle being solved, so player 0 wins.

The winning strategy for player 2 when the algorithm returns False is similar.

Efficiency: We've only claimed that this algorithm takes polynomial space. Storing the values of mn moves takes $O(mn)$ space, and everything else takes $O(1)$ space. (This algorithm happens to have exponential runtime.) □

6 QSAT and 2-player bounded games.

Almost nothing in our proof that JigsawA2 is in PSPACE depended on the details of what JigsawA2 is. In fact, the only facts we used were that:

1. If we know the complete history of the game, we can tell whether player 0 won.
2. We can list all (polynomially many) possibilities for each move.
3. We have a polynomial bound on the length of the game.

So, the same argument proves that *every* game with the same properties is in PSPACE. For instance, we can turn SAT into a game by letting the players take turns choosing a value for each variable (in order), where player 0 wins if the SAT formula ends up satisfied. Deciding whether player 0 has a winning strategy in that game is called "Quantified Satisfiability" (QSAT)[1], and the same argument shows that it's in PSPACE.

[1] QSAT was historically called "True Quantified Boolean Formula" (TQBF).

Input	: A boolean formula φ on n variables x_0, x_1, ..., x_{n-1}
Output	: True iff $\exists x_0 \forall x_1 \forall x_2 \exists x_3 \cdots : \varphi(x_0, \ldots, x_{n-1})$. False otherwise.

Computational Problem QuantifiedSatisfiability

QSAT is the canonical example of a PSPACE-complete problem, although proving that QSAT is PSPACE-hard is out of the scope of CS 120.

As another example, those properties are true of chess, even if we generalize it to an $n \times n$ board:

1. If we know the complete history of the game, we can reconstruct the final position and see whether player 1's king is checkmated.

2. The board has only polynomially many pieces on it, each with polynomially many possible moves.

3. Chess has a polynomial bound on the number of moves, because if 50 moves pass without any pawns moving or pieces being captured, the game ends. Pawns can be advanced only polynomially many times before they all reach the end of the board, and pieces can be captured at most polynomially many times before there are none left to capture, making the length of the game at most polynomial.

In fact, deciding whether white wins generalized chess (with that 50-move rule) is PSPACE-hard as well, making it PSPACE-complete.

7 Unbounded-time games

Our proof that chess is in PSPACE relied on the 50-move rule. That rule was created in the belief that it would never end games that it's possible for one player to win, only save time by ending games where neither player could make progress. However, in the last few decades, it's been proved that there are chess positions which one player could win without the 50-move rule, but can't win before the 50-move rule ends the game. What if we got rid of that rule?

Even without the 50-move rule, we need only polynomially much memory to describe the state of a chess board, so one might hope that solving the game would still be in PSPACE. Our previous algorithm can no longer get by with polynomially much space, since it stores an entire history of the game, which might be of more than polynomial size.

In fact, chess without the 50-move rule turns out to be EXP-complete. Modern computer scientists believe (but have not proven) that there are problems solvable in exponential time but not solvable in polynomial space, so chess without the 50-move rule is probably not in PSPACE.

We could similarly make a variant of the 2-player jigsaw game without a polynomial bound on the length of the game: for instance, say that on a player's turn, a player can *either* place their next piece adjacent to a previously one *or move one of their previously-placed pieces* to an adjacent legal position. That game, called JigsawB2, is also in EXP, and is probably EXP-complete (although, as far as the instructors are aware, no one has worked through the details to prove so).

8 Unbounded-space games

There's one final natural way we could extend the problem of jigsaw puzzles: make the puzzle itself infinite. That is, define an infinite jigsaw as follows:

Definition 8.1. An *infinite jigsaw puzzle* consists of a set $S = \{t_0, t_1, \ldots\}$ of "jigsaw tiles".
A *jigsaw tile* consists of four integers t.left, t.right, t.top, and t.bottom.
A *jigsaw partial solution* is a function $f : S \to (\mathbb{Z} \times \mathbb{Z}) \cup \{\text{"unassigned"}\}$ such that:

1. For all $t, t' \in S$, $f(t) \neq f(t')$ or $f(t) =$ "unassigned".
2. For all $t, t' \in S$, if $f(t) = f(t') + (0, 1)$, then f(t).top = -f(t').bottom.
3. For all $t, t' \in S$, if $f(t) = f(t') + (1, 0)$, then f(t).right = -f(t').left.

A *jigsaw solution* is a jigsaw partial solution in which no piece is "unassigned".

Input	**:** A jigsaw puzzle, as defined above.
Output	**:** A jigsaw solution, as defined above.

Computational Problem JigsawC1

This problem is undecidable, but we won't have time to prove it in class.