1 E.A.5.4

1.1 Node consistency

```rust
// self è l'istanza di un CSP
pub fn make_node_consistent(&mut self) -> bool {
    // Assegnamento vuoto usato per controllare i vincoli
    let mut assignment = vec![None; self.domains.len()];

    // I vincoli unari sono separati dal resto
    for Constraint(vars, c) in self.unary.iter() {
        // Sola variabile coinvolta nel vincolo
        let var = vars[0];
        // Valori rimossi dal dominio di `var`
        let mut removed = vec![];

        // Dato che il vincolo è una funzione del tipo
        // Assignment -> bool
        // l'unico modo per controllare se un valore soddisfa
        // il vincolo è assegnarlo e controllare il risultato
        for &val in self.domains[var].iter() {
            assignment[var] = Some(val);
            if !c(&assignment) {
                removed.push(val);
            }
        }

        // Se tutti i valori del dominio sono stati rimossi
        // non c'è una soluzione al CSP
        if self.domains[var].len() == removed.len() {
            return false;
        }

        for val in removed {
            self.domains[var].remove(&val);
        }

        assignment[var] = None;
    }

    // C'è una possibile soluzione al CSP
    true
}
```

Questa è un'implementazione un po' sempliciotta, perché si potrebbero ridurre i domini in modo più efficiente se fosse noto il tipo di vincolo specifico (ad esempio, un vincolo $X_3 = 5$ si potrebbe gestire senza dover controllare tutti i valori del dominio). Vedere `src/csp.rs` per più dettagli sulla definizione di `struct Constraint` e `struct CSP`.

Al termine della chiamata a `make_node_consistent()` non c'è nessun dominio vuoto, e il dominio di X_5 è stato ridotto a $\{3, 4, 5\}$.

1.2 GAC

I risultati di GAC-3

```
X1 > X3, viene rimosso 1 dal dominio di X1
[
  {2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5},
  {1, 2, 3, 4, 5}, {3, 4, 5}
]

X1 > X3, viene rimosso 5 dal dominio di X3
[
  {2, 3, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4},
  {1, 2, 3, 4, 5}, {3, 4, 5}
]

X2 ≤ X3, viene rimosso 5 dal dominio di X2
[
  {2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3, 4},
  {1, 2, 3, 4, 5}, {3, 4, 5}
]

X2 ≤ X3, non cambia nulla
[
  {2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3, 4},
  {1, 2, 3, 4, 5}, {3, 4, 5}
]

X3^2 + X4^2 ≤ 15, viene rimosso 4 dal dominio di X3
[
  {2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3},
  {1, 2, 3, 4, 5}, {3, 4, 5}
]

X3^2 + X4^2 ≤ 15, viene rimosso 4 dal dominio di X4
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X1 + X5 ≥ 3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X1 + X5 ≥ 3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X1 > X3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X1 > X3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]
```

```
X2 ≤ X3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X2 ≤ X3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]

X2 ≤ X3, non cambia nulla
[{2, 3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3}, {1, 2, 3}, {3, 4, 5}]
```

Nota: questo è il peggior codice che ho scritto nell'ultimo periodo, ma a corto di tempo ci si accontenta di far funzionare le cose.

```rust
pub fn gac_3(&mut self) -> bool {
    let mut queue = VecDeque::new();

    for c in self.general.iter() {
        for &var in c.0.iter() {
            queue.push_back((var, c));
        }
    }

    while let Some((var, c)) = queue.pop_front() {
        let mut removed = vec![];

        let other_vars: Vec<_> = c.0.iter()
            .filter(|&&var_2| var_2 != var).collect();
        let mut assignment = vec![None; self.domains.len()];
        for &val in self.domains[var].iter() {
            assignment[var] = Some(val);
            for &&other_var in other_vars.iter() {
                assignment[other_var] =
                    self.domains[other_var].first().copied();
            }

            let mut satisfied = false;

            for combination in
                self.combinations(
                    other_vars.clone()
                        .into_iter().copied().collect()
                )
            {
                for (var, val_2) in combination {
                    assignment[var] = Some(val_2);
                }

                if c.1(&assignment) {
                    satisfied = true;
                    break;
```

```
                }
            }

            if !satisfied {
                removed.push(val);
            }
        }

        for val in removed.iter() {
            self.domains[var].remove(val);
        }

        if !removed.is_empty() {
            if self.domains[var].is_empty() {
                return false;
            }

            for &other_var in other_vars {
                queue.push_back((other_var, c));
            }
        }
    }

    true
}

fn combinations(
    &self, vars: Vec<usize>
) -> Vec<Vec<(usize, usize)>> {
    if vars.is_empty() {
        vec![vec![]]
    } else {
        let mut combinations = vec![];

        let mut rest = vars.clone();
        let last = rest.pop().unwrap();

        for combination in self.combinations(rest) {
            for &val in self.domains[last].iter() {
                let mut c = combination.clone();
                c.push((last, val));
                combinations.push(c);
            }
        }

        combinations
    }
}
```