CSP-Solver

Generated by Doxygen 1.8.17

1 CSP-Solver

Project for the lecture "Objektorientierte Programmierung mit C++" at Ulm University. Solves arbitrary binary constraint satisfaction problems (CSP) using the `AC-3 algorithm` and backtracking search. The seminar paper (German) for this project can be found `here`.

1.1 Doxygen Documentation

- `HTML`
- `PDF`

1.2 How To

A CPS consists of a set of variables, represented by `csp::Variable` and a set of constraints (dependencies between pairs of variables) represented by `csp::Constraint` or `csp::Arc`. You can use your own variable type by deriving from `csp::Variable`.

1.2.1 Creating Variables

To create a variable, a domain of values has to be given, representing all possible values the variable might take:
```
#include "Variable.h"
using MyVar = csp::Variable<int>;
MyVar a({1, 2, 17, 24});
```

Also possible: Create your own type. This allows you to add functionality to your variable type. For example, you can add a variable name:
```
#include <string>
#include "Variable.h"
class MyVar : public csp::Variable<int> {
public:
  explicit MyVar(std::string name) : csp::Variable<int>({1, 2, 3, 4}), name(std::move(name)) {}
  const std::string name;
}
```

1.2.2 Specifying Constraints

`csp::Constraint` or `csp::Arc` specify dependencies between pairs of variables. They contain a pointer type to each variable and a binary predicate specifying the constraint. You can use arbitrary pointer types that support the dereference-operator as well as -> operator. Example using the custom variable type above and shared_ptr (which I recommend over raw pointers):
```
#include <memory>
#include "Arc.h"
auto varA = std::make_shared<MyVar>("A");
auto varB = std::make_shared<MyVar>("B");
csp::Constraint aLessB(varA, varB, std::less<>());
```

You can also use `csp::Arc` to specify the relation between variables. The difference is, that an arc describes a directed constraint. Even if $A < B$ is equivalent to $B > A$, two arcs describing both relations respectively are not. When defining your CSP using arcs, make sure that always both directions are specified explicitly. When using `csp::Constraint` only one direction suffices. In some situations, it is easier to specify the CSP using arcs than using constraints or vice versa. You cannot mix arcs and constraints when creating a CSP but you can convert a `csp::Constraint` to two equivalent `csp::Arc`.

1.2.3 Creating the CSP

Once you specified all variables and the respective constraints, create your CSP using:
```
csp::Csp myCsp = csp::make_csp(std::array{varA, varB}, std::array{aLessB});
```

You can use arbitrary containers that support iteration.

1.2.4 Solving the CSP

An instance of `csp::Csp` can be solved using:
```
bool success = csp::solve(myCsp);
```

If solving the CSP is possible all domains of all variables will be reduced to exactly one value. The algorithm will find a solution if one exists (given engough time). If multiple exist, it is unspecified which exact solution is found. If the CSP cannot be solved, the function returns `false` but might still modify the variables' value domains.

1.2.4.1 Specifying a Solving Strategy

By default, `csp::solve` uses the minimum remaining values strategy, meaning that the algorithm chooses the variable with the fewest remaining values in its domain to be assigned next. You can also use a different (even custom) strategy e.g.:
```
auto strat = [](const auto & problem) {
  // Your code here -> return the desired unassigned variable from the CSP
  return theNextVar;
};
bool success = csp::solve(myCsp, strat);
```

Examples on how to create your own strategy can be found in the file `src/strategies.h`.

1.3 Solving Sudoku Puzzles

The `src/main.cpp` contains a program that can solve Sudoku puzzles. A Sudoku is defined by a grid of numbers where a 0 indicates, that the respective field is yet to be assigned. Some examples are provided in the `res` directory.

2 Namespace Documentation

2.1 csp Namespace Reference

Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.

Namespaces

- strategies

    contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.
- util

    contains utility functions used by the search algorithm

Data Structures

- class Arc
- class Constraint
- struct Csp
- class Variable

Typedefs

- template<typename T >
  using **BinaryPredicate** = std::function< bool(const T &, const T &)>

Functions

- template<typename VarPtr , typename Strategy >
  bool recursiveSolve (Csp< VarPtr > &problem, const Strategy &strategy)
- template<typename VarPtr , typename Strategy = strategies::Mrv<VarPtr>>
  bool solve (Csp< VarPtr > &problem, const Strategy &strategy=Strategy())
- template<typename VarContainer , typename ArcContainer , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(*std::begin(std::declval< ArcContainer >()))>>::value , int >
  auto make_csp (const VarContainer &variables, const ArcContainer &arcs) -> Csp< std::decay_t< decltype(std::end(arcs), std::end(variables), *std::begin(variables))>>
- template<typename VarContainer , typename ContraintContainer , std::enable_if_t< type_traits::is_constraint< std::remove_reference_t< decltype(*std::begin(std::declval< ContraintContainer >()))>>::value , int >
  auto make_csp (const VarContainer &variables, const ContraintContainer &constraints) -> Csp< std::decay_t< decltype(std::end(constraints), std::end(variables), *std::begin(variables))>>
- template<typename VarIt , typename ConstrIt , std::enable_if_t< type_traits::is_constraint< std::remove_reference_t< decltype(*std::declval< ConstrIt >())>>::value , int >
  auto make_csp (VarIt vBegin, VarIt vEnd, ConstrIt cBegin, ConstrIt cEnd) -> Csp< std::decay_t< decltype(*++cBegin, cBegin==cEnd,++vBegin, vBegin==vEnd, *vBegin)>>
- template<typename VarIt , typename ArcIt , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(*std::declval< ArcIt >())>>::value , int >
  auto make_csp (VarIt vBegin, VarIt vEnd, ArcIt aBegin, ArcIt aEnd) -> Csp< std::decay_t< decltype(*++aBegin, aBegin==aEnd,++vBegin, vBegin==vEnd, *vBegin)>>

2.1.1 Detailed Description

Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.

2.1.2 Function Documentation

2.1.2.1 make_csp() [1/4]

```
template<typename VarContainer , typename ArcContainer , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(*std::begin(std::declval< ArcContainer >()))>>::value , int >
auto csp::make_csp (
            const VarContainer & variables,
            const ArcContainer & arcs ) -> Csp<std::decay_t<decltype( std::end(arcs), std::end(variables), *std::begin(variables) )>>
```

Creates a CSP from a container of variable-pointer and a container of csp::Arcs

Template Parameters

VarContainer	Container-Type containing pointer-types to a type derived of csp::Variable
ArcContainer	Container-Type containing csp::Arcs

Parameters

variables	Container of all variables in the CSP
arcs	Container of all directed csp::Arcs in the CSP

Returns

csp::Csp representing the problem induced by the given variables and arcs

Note

When using csp::Arcs to specify the constraints, make sure that if you have a constraint e.g. A $<$ B, you specify both csp::Arcs representing A $<$ B and B $>$ A! Otherwise the problem is malformed and may lead to invalid solutions!

2.1.2.2 make_csp() [2/4]

```
template<typename VarContainer , typename ContraintContainer , std::enable_if_t< type_traits::is_constraint< std::remove_reference_t< decltype(*std::begin(std::declval< ContraintContainer >()))>>::value , int >
auto csp::make_csp (
            const VarContainer & variables,
            const ContraintContainer & constraints ) -> Csp<std::decay_t<decltype( std::end(constraints), std::end(variables), *std::begin(variables) )>>
```

Creates a CSP from a container of variable-pointers and a container of csp::Constraints

Template Parameters

VarContainer	Container-Type containing pointer-types to a type derived of csp::Variable
ArcContainer	Container-Type containing csp::Constraints

Parameters

variables	Container of all variables in the CSP
arcs	Container of all undirected csp::Constraints in the CSP

Returns

csp::Csp representing the problem induced by the given variables and constraints

Note

When using csp::Constraints to specify the constraints, specify them only once. A csp::Constraint for e.g. A < B fully represents the constraint between the csp::Variable A and B. Specifying A < B and B > A may lead to performance loss during search!

2.1.2.3 make_csp() [3/4]

```
template<typename VarIt , typename ArcIt , std::enable_if_t< type_←
traits::is_arc< std::remove_reference_t< decltype(*std::declval< ArcIt >())>>::value , int >
auto csp::make_csp (
            VarIt vBegin,
            VarIt vEnd,
            ArcIt aBegin,
            ArcIt aEnd ) -> Csp<std::decay_t<decltype(*++aBegin, aBegin == aEnd, ++vBegin,
vBegin == vEnd, *vBegin)>>
```

Creates a CSP from a container of variable-pointers and a container of csp::Arc using iterators. Variables and arcs are taken from the respective range [begin, end)

Template Parameters

Var←It	Iterator type of variable container
Arc←It	Iterator type of arc container

Parameters

vBegin	start of range of variables
vEnd	end of range of variables (exclusive)
aBegin	start of range of arcs
aEnd	start of range of arcs (exclusive)

Returns

csp::Csp representing the problem induced by the given variables and arcs

Note

When using csp::Arcs to specify the constraints, make sure that if you have a constraint e.g. A $<$ B, you specify both csp::Arcs representing A $<$ B and B $>$ A! Otherwise the problem is malformed and may lead to invalid solutions!

2.1.2.4 make_csp() [4/4]

```
template<typename VarIt , typename ConstrIt , std::enable_if_t<
type_traits::is_constraint< std::remove_reference_t< decltype(*std::declval< ConstrIt >())>>↩
::value , int >
auto csp::make_csp (
            VarIt vBegin,
            VarIt vEnd,
            ConstrIt cBegin,
            ConstrIt cEnd ) -> Csp<std::decay_t<decltype(*++cBegin, cBegin == cEnd, ++v↩
Begin, vBegin == vEnd, *vBegin)>>
```

Creates a CSP from a container of variable-pointers and a container of csp::Constraint using iterators. Variables and constraints are taken from the respective range [begin, end)

Template Parameters

VarIt	Iterator type of variable container
Constr↩ It	Iterator type of constraint container

Parameters

vBegin	start of range of variables
vEnd	end of range of variables (exclusive)
cBegin	start of range of constraints
cEnd	start of range of constraints (exclusive)

Returns

csp::Csp representing the problem induced by the given variables and constraints

Note

When using csp::Constraints to specify the constraints, specify them only once. A csp::Constraint for e.g. A $<$ B fully represents the constraint between the csp::Variable A and B. Specifying A $<$ B and B $>$ A may lead to performance loss during search!

2.1.2.5 recursiveSolve() `template<typename VarPtr , typename Strategy >`

```
bool csp::recursiveSolve (
            Csp< VarPtr > & problem,
            const Strategy & strategy )
```

Recursive backtracking search for csp::Csps. Prefer using the wrapper function csp::solve

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable
Strategy	Type of value selection strategy during search

Parameters

problem	CSP to be solved
strategy	value selection strategy object used during during search

Returns

True if problem was solved, false otherwise

2.1.2.6 solve() `template<typename VarPtr , typename Strategy = strategies::Mrv<VarPtr>>`

```
bool csp::solve (
            Csp< VarPtr > & problem,
            const Strategy & strategy = Strategy() )
```

Solves a CSP. If a solution exists, the value domains of each variable in the given Csp will be reduced to exactly one value. If multiple solutions exist, it is unspecified which is found. If no solution exists, false is returned but the value domains of the variables might still be altered

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable
Strategy	Type of value selection strategy during search (default: minimum remaining values strategy). Has to provide ()-Operator and return VarPtr from given csp::Csp

Parameters

problem	CSP to be solved

Returns

True if problem was solved, false otherwise

2.2 csp::strategies Namespace Reference

contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.

Data Structures

- struct First
- struct Mrv

2.2.1 Detailed Description

contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.

2.3 csp::util Namespace Reference

contains utility functions used by the search algorithm

Typedefs

- template< typename VarT >
  using **CspCheckpoint** = std::vector< typename VarT::DomainT >

Functions

- template< typename VarPtr >
  bool removeInconsistent (const Arc< VarPtr > &arc)
- template< typename VarPtr >
  bool ac3 (Csp< VarPtr > &problem)
- template< typename VarPtr >
  auto makeCspCheckpoint (const Csp< VarPtr > &problem) -> CspCheckpoint< typename Csp< VarPtr >::VarT >
- template< typename VarPtr >
  void restoreCspFromCheckpoint (Csp< VarPtr > &problem, const CspCheckpoint< typename Csp< VarPtr >::VarT > &checkpoint)

2.3.1 Detailed Description

contains utility functions used by the search algorithm

2.3.2 Function Documentation

2.3.2.1 ac3() `template<typename VarPtr >`

```
bool csp::util::ac3 (
            Csp< VarPtr > & problem )
```

Obtains arc consistency in a CSP using the AC3-algorithm

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

Parameters

problem	The CSP to be processed

Returns

True if arc consistency was obtained, false if not possible

2.3.2.2 makeCspCheckpoint()

```
template<typename VarPtr >
auto csp::util::makeCspCheckpoint (
            const Csp< VarPtr > & problem ) -> CspCheckpoint<typename Csp<VarPtr>::VarT>
```

Backs up all value domains of all variables in a CSP

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

Parameters

problem	The CSP to be backed up

Returns

vector of csp::Variable domains. Domains are ordered according to the variables in the CSP

2.3.2.3 removeInconsistent()

```
template<typename VarPtr >
bool csp::util::removeInconsistent (
            const Arc< VarPtr > & arc )
```

Removes all inconsistent values from the source node of the given csp::Arc. As a result, the source node's value domain only contains values for which a valid value in the domain of the destination node exists.

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

Parameters

arc	Arc to be processed

Returns

True if the value domain of the source node was modified, false otherwise

2.3.2.4 restoreCspFromCheckpoint()

```
template<typename VarPtr >
void csp::util::restoreCspFromCheckpoint (
            Csp< VarPtr > & problem,
            const CspCheckpoint< typename Csp< VarPtr >::VarT > & checkpoint )
```

Restores the value domains of all csp::Variables in a CSP from the given checkpoint

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

Parameters

problem	The CPS to be restored
checkpoint	Checkpoint to load the value domains from

3 Data Structure Documentation

3.1 csp::Arc< VarPtr > Class Template Reference

```
#include <Arc.h>
```

Public Types

- using **VarType** = typename Constraint< VarPtr >::VarType

Public Member Functions

- Arc (VarPtr v1, VarPtr v2, BinaryPredicate< VarType > predicate, bool reverse=false) noexcept(Constraint< VarPtr >::nothrow_construcible)
- constexpr void reverse () noexcept
- constexpr VarPtr from () const noexcept
- constexpr VarPtr to () const noexcept
- bool constraintSatisfied (const VarType &valFrom, const VarType &valTo) const

3.1.1 Detailed Description

template<typename VarPtr>
class csp::Arc< VarPtr >

Represents a binary constraint as directed arc in a constraint satisfaction problem. Is mainly used during solving to obtain arc consistency

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

Note

Constraints implicitly specify two directed arcs. For example: A constraint A < B is equivalent to two arcs A < B and B > A

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Arc() template<typename VarPtr >

```
csp::Arc< VarPtr >::Arc (
        VarPtr v1,
        VarPtr v2,
        BinaryPredicate< VarType > predicate,
        bool reverse = false )  [inline], [noexcept]
```

Ctor

Parameters

v1	Pointer-Type to first variable
v2	Pointer-Type to first variable
predicate	Constraint in form of a binary predicate
reverse	Specifies whether the arc represents v2 -> v1 instead

3.1.3 Member Function Documentation

3.1.3.1 constraintSatisfied() template<typename VarPtr >

```
bool csp::Arc< VarPtr >::constraintSatisfied (
        const VarType & valFrom,
        const VarType & valTo ) const  [inline]
```

Checks if the binary constraint between source and destination is satisfied. If values are chosen from the domains of from() and to() respectively, makes sure the constraint predicate is evaluated correctly

Parameters

valFrom	value of the source node
valTo	value of the destination node

Returns

true if constraint is satisfied, false otherwise

3.1.3.2 from() `template<typename VarPtr >`

`constexpr VarPtr csp::Arc< VarPtr >::from ( ) const [inline], [constexpr], [noexcept]`

Gets the source node of the arc

Returns

Always returns the pointer to the source node of the arc, taking into account if the arc is reversed

3.1.3.3 reverse() `template<typename VarPtr >`

`constexpr void csp::Arc< VarPtr >::reverse ( ) [inline], [constexpr], [noexcept]`

Reverses the arc (switches from() and to() members)

3.1.3.4 to() `template<typename VarPtr >`

`constexpr VarPtr csp::Arc< VarPtr >::to ( ) const [inline], [constexpr], [noexcept]`

Gets the destination node of the arc

Returns

Always returns the pointer to the destination node of the arc, taking into account if the arc is reversed

The documentation for this class was generated from the following file:

- src/Arc.h

3.2 csp::Constraint< VarPtr > Class Template Reference

```
#include <Arc.h>
```

Public Types

- using **VarType** = typename std::remove_reference_t< decltype(∗std::declval< VarPtr >())>::ValueT
- using **ArcT** = Arc< VarPtr >

Public Member Functions

- Constraint (VarPtr v1, VarPtr v2, BinaryPredicate< VarType > predicate) noexcept(nothrow_construcible)
- auto getArcs () const -> std::pair< ArcT, ArcT >

Protected Attributes

- VarPtr **var1**
- VarPtr **var2**
- BinaryPredicate< VarType > **predicate**

Static Protected Attributes

- static constexpr bool **nothrow_construcible**

3.2.1 Detailed Description

template<typename VarPtr>
class csp::Constraint< VarPtr >

Represents a binary undirected constraint in a constraint satisfaction problem

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Constraint() `template<typename VarPtr >`

```
csp::Constraint< VarPtr >::Constraint (
            VarPtr v1,
            VarPtr v2,
            BinaryPredicate< VarType > predicate )  [inline], [noexcept]
```

Ctor

Parameters

v1	Pointer-Type to first variable
v2	Pointer-Type to first variable
predicate	Constraint in form of a binary predicate

3.2.3 Member Function Documentation

3.2.3.1 getArcs() `template<typename VarPtr >`

```
auto csp::Constraint< VarPtr >::getArcs ( ) const -> std::pair<ArcT, ArcT>  [inline]
```

Create the two equivalent directed csp::Arcs

Returns

Pair of equivalent csp::Arcs

3.2.4 Field Documentation

3.2.4.1 nothrow_construcible template<typename VarPtr >

constexpr bool csp::Constraint< VarPtr >::nothrow_construcible [static], [constexpr], [protected]

Initial value:

```
= std::is_nothrow_move_constructible_v<VarPtr> &&
                                                        std::is_nothrow_move_constructible_v<BinaryPredicate <
       VarType»
```

The documentation for this class was generated from the following file:

- src/Arc.h

3.3 csp::Csp< VarPtr > Struct Template Reference

`#include <Csp.h>`

Public Types

- using **ArcT** = Arc< VarPtr >
- using **VarT** = std::remove_reference_t< decltype(∗std::declval< VarPtr >())>
- using **VarListT** = std::vector< VarPtr >
- using **ArcListT** = std::deque< ArcT >
- using **NeighbourListT** = std::unordered_map< VarPtr, std::vector< ArcT > >

Data Fields

- const VarListT **variables**
- const ArcListT **arcs**
- const NeighbourListT **incomingNeighbours**

Friends

- template<typename VarIt , typename ArcIt , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(∗std::declval< ArcIt >())>>::value >
  auto make_csp (VarIt vBegin, VarIt vEnd, ArcIt aBegin, ArcIt aEnd) -> Csp< std::decay_t< decltype(∗++a↩Begin, aBegin==aEnd,++vBegin, vBegin==vEnd, ∗vBegin)>>

3.3.1 Detailed Description

template<typename VarPtr>
struct csp::Csp< VarPtr >

Represents a constraint satisfaction problem (CSP)

Template Parameters

VarPtr	VarPtr Pointer-type to a type derived from csp::Variable

3.3.2 Friends And Related Function Documentation

3.3.2.1 make_csp `template<typename VarPtr >`

```
template<typename VarIt , typename ArcIt , std::enable_if_t< type_traits::is_arc< std::remove↩
_reference_t< decltype(*std::declval< ArcIt >())>>::value >
auto make_csp (
            VarIt vBegin,
            VarIt vEnd,
            ArcIt aBegin,
            ArcIt aEnd ) -> Csp<std::decay_t<decltype(*++aBegin, aBegin == aEnd, ++vBegin,
vBegin == vEnd, *vBegin)>>  [friend]
```

Creates a CSP from a container of variable-pointers and a container of csp::Arc using iterators. Variables and arcs are taken from the respective range [begin, end)

Template Parameters

VarIt	Iterator type of variable container
ArcIt	Iterator type of arc container

Parameters

vBegin	start of range of variables
vEnd	end of range of variables (exclusive)
aBegin	start of range of arcs
aEnd	start of range of arcs (exclusive)

Returns

csp::Csp representing the problem induced by the given variables and arcs

Note

When using csp::Arcs to specify the constraints, make sure that if you have a constraint e.g. A < B, you specify both csp::Arcs representing A < B and B > A! Otherwise the problem is malformed and may lead to invalid solutions!

The documentation for this struct was generated from the following file:

- src/Csp.h

3.4 csp::strategies::First< VarPtr > Struct Template Reference

```
#include <strategies.h>
```

Public Member Functions

- VarPtr **operator()** (const Csp< VarPtr > &problem) const

3.4.1 Detailed Description

template<typename VarPtr>
struct csp::strategies::First< VarPtr >

Variable selection strategy that simply chooses the next unassigned variable

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

The documentation for this struct was generated from the following file:

- src/strategies.h

3.5 csp::type_traits::is_arc< T > Struct Template Reference

```
#include <Csp.h>
```

Inherits declval< T >.

3.5.1 Detailed Description

template<typename T>
struct csp::type_traits::is_arc< T >

Used to check if type is of template type csp::Arc

Template Parameters

T	Type to be checked

The documentation for this struct was generated from the following file:

- src/Csp.h

3.6 csp::type_traits::is_constraint< T > Struct Template Reference

```
#include <Csp.h>
```

Static Public Attributes

- static constexpr bool **value**

3.6.1 Detailed Description

template<typename T>
struct csp::type_traits::is_constraint< T >

Used to check if type is of template type csp::Constraint

Template Parameters

T	Type to be checked

3.6.2 Field Documentation

3.6.2.1 value `template<typename T >`

`constexpr bool csp::type_traits::is_constraint< T >::value [static], [constexpr]`

Initial value:

```
= decltype(implementations::constraintTest(std::declval<T>()))::value
                    && ! is_arc<T>::value
```

The documentation for this struct was generated from the following file:

- src/Csp.h

3.7 csp::type_traits::is_dereferencable< T > Struct Template Reference

```
#include <Arc.h>
```

Inherits declval< T >.

3.7.1 Detailed Description

template<typename T>
struct csp::type_traits::is_dereferencable< T >

Used to check if type can be dereferenced using ∗-Operator

Template Parameters

T	Type to be checked

The documentation for this struct was generated from the following file:

- src/Arc.h

3.8 csp::type_traits::is_derived_from_var< T > Struct Template Reference

```
#include <Arc.h>
```

Inherits declval< std::remove_reference_t< T > * >.

3.8.1 Detailed Description

template<typename T>
struct csp::type_traits::is_derived_from_var< T >

Used to check if type is derived from csp::Variable

Template Parameters

T	Type to be checked

The documentation for this struct was generated from the following file:

- src/Arc.h

3.9 csp::strategies::Mrv< VarPtr > Struct Template Reference

```
#include <strategies.h>
```

Public Member Functions

- VarPtr **operator()** (const Csp< VarPtr > &problem) const

3.9.1 Detailed Description

template<typename VarPtr>
struct csp::strategies::Mrv< VarPtr >

Minimum remaining values strategy for variable selection during search. Chooses the variable with the fewest remaining possible values

Template Parameters

VarPtr	Pointer-type to a type derived from csp::Variable

The documentation for this struct was generated from the following file:

- src/strategies.h

3.10 csp::Variable< T, DomainType > Class Template Reference

```
#include <Variable.h>
```

Inherited by SudokuNode.

Public Types

- using **DomainT** = DomainType< T >
- using **ValueT** = T

Public Member Functions

- Variable (DomainT domain)
- Variable (std::initializer_list< T > init)
- void assign (T val)
- constexpr bool isAssigned () const noexcept(noexcept(std::declval< DomainT >().size()))
- template<typename Container , std::enable_if_t< std::is_convertible_v< decltype(*std::begin(std::declval< Container >())), T > , int >
  void setValueDomain (const Container &container)
- void setValueDomain (DomainT values)
- constexpr auto valueDomain () const noexcept -> const DomainT &
- constexpr auto valueDomain () noexcept -> DomainT &

3.10.1 Detailed Description

template<typename T, template< typename... > typename DomainType = std::list>
class csp::Variable< T, DomainType >

Represents a variable in a constraint satisfaction problem

Template Parameters

T	Type of the contained value
DomainType	Type of container used for the variables domain (default is std::list)

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Variable() [1/2]

```
template<typename T , template< typename... > typename DomainType = std::list>
csp::Variable< T, DomainType >::Variable (
            DomainT domain ) [inline], [explicit]
```

CTor

Parameters

domain	list of possible values

3.10.2.2 Variable() [2/2]

```
template<typename T , template< typename... > typename DomainType = std::list>
csp::Variable< T, DomainType >::Variable (
            std::initializer_list< T > init ) [inline]
```

Ctor using initializer list

Parameters

init	

3.10.3 Member Function Documentation

3.10.3.1 assign()

```
template<typename T , template< typename... > typename DomainType = std↩
::list>
void csp::Variable< T, DomainType >::assign (
            T val ) [inline]
```

Sets the variable to the specified value (by reducing the value domain to said value)

Parameters

val	The desired value

Note

The value does not have to be in the variable's value domain. No checks are performed.

3.10.3.2 isAssigned()

```
template<typename T , template< typename... > typename DomainType = std::list>
constexpr bool csp::Variable< T, DomainType >::isAssigned ( ) const [inline], [constexpr], [noexcept]
```

Checks if the variable is assigned

Returns

True if the variable's domain contains exactly one value, false otherwise

3.10.3.3 setValueDomain() [1/2]

```
template<typename T , template< typename... > typename DomainType = std::list>
template<typename Container , std::enable_if_t< std::is_convertible_v< decltype(*std::begin(std::declval< Container >())), T > , int >
void csp::Variable< T, DomainType >::setValueDomain (
            const Container & container ) [inline]
```

Sets the value domain accepting arbitrary containers that support iteration

Template Parameters

Container	Type of the container

Parameters

container	desired values

Note

Avoid duplicates! No checks are performed. Duplicates can lead to performance losses.

3.10.3.4 setValueDomain() [2/2]

```
template<typename T , template< typename... > typename DomainType = std::list>
void csp::Variable< T, DomainType >::setValueDomain (
            DomainT values ) [inline]
```

Sets the value domain

Parameters

values	desired values.

Note

Avoid duplicates! No checks are performed. Duplicates can lead to performance losses.

3.10.3.5 valueDomain() [1/2]

```
template<typename T , template< typename... > typename Domain↩
Type = std::list>
constexpr auto csp::Variable< T, DomainType >::valueDomain ( ) const -> const DomainT&   [inline],
[constexpr], [noexcept]
```

Gets the value domain

Returns

3.10.3.6 valueDomain() [2/2]

```
template<typename T , template< typename... > typename Domain↩
Type = std::list>
constexpr auto csp::Variable< T, DomainType >::valueDomain ( ) -> DomainT &   [inline], [constexpr],
[noexcept]
```

Gets the value domain

Returns

The documentation for this class was generated from the following file:

- src/Variable.h

4 File Documentation

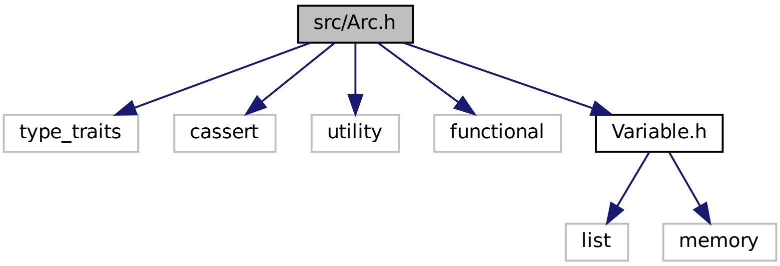
4.1 src/Arc.h File Reference

This file contains the the csp::Arc and the csp::Constraint class. Both can be used to specify constraints (dependencies) between two csp::Variable pointers. An Arc is directed (specifying a constraint that a Variable X X imposes on a Variable Y, e.g. A < B but not B > A). A Constraint implicitly defines both directions (e.g. if A < B <=> B > A). The actual constraint is given as a binary predicate.

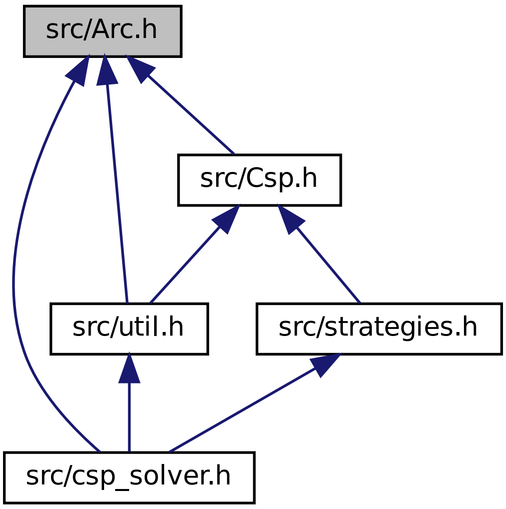
```
#include <type_traits>
#include <cassert>
#include <utility>
#include <functional>
```

`#include "Variable.h"`

Include dependency graph for Arc.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct csp::type_traits::is_dereferencable< T >
- struct csp::type_traits::is_derived_from_var< T >
- class csp::Arc< VarPtr >
- class csp::Constraint< VarPtr >
- class csp::Arc< VarPtr >

Namespaces

- csp

  Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.

Typedefs

- template<typename T >
  using **csp::BinaryPredicate** = std::function< bool(const T &, const T &)>

Functions

- template<typename T >
  std::true_type **csp::type_traits::implementations::pointerTest** (decltype(∗std::declval< T >(), std↩
  ::declval< T >()))
- template<typename T >
  std::false_type **csp::type_traits::implementations::pointerTest** (...)
- template<typename VarType , template< typename... >typename Domain>
  std::true_type **csp::type_traits::implementations::derivedTest** (Variable< VarType, Domain > ∗)
- std::false_type **csp::type_traits::implementations::derivedTest** (...)

4.1.1 Detailed Description

This file contains the the csp::Arc and the csp::Constraint class. Both can be used to specify constraints (dependencies) between two csp::Variable pointers. An Arc is directed (specifying a constraint that a Variable X X imposes on a Variable Y, e.g. A < B but not B > A). A Constraint implicitly defines both directions (e.g. if A < B <=> B > A). The actual constraint is given as a binary predicate.

Author

Tim Luchterhand

Date

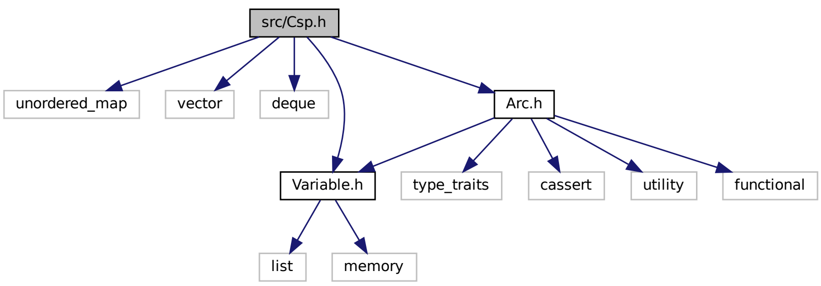
22.07.20

4.2 src/Csp.h File Reference

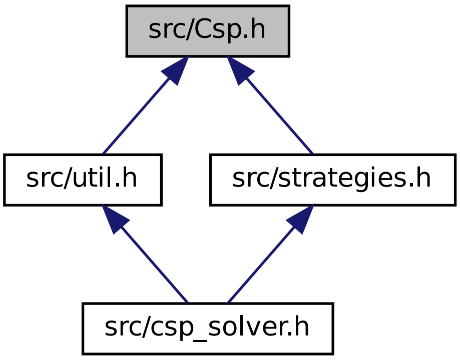
This file contains the csp::Csp class which represents a complete definition of a constraint satisfaction problem. The class contains a list of Variable pointers, a list of all arcs and a map that specifies incoming arcs of each variable. The csp::Csp should be created using the provided function csp::make_csp.

```
#include <unordered_map>
#include <vector>
#include <deque>
#include "Variable.h"
#include "Arc.h"
```

Include dependency graph for Csp.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct csp::type_traits::is_arc< T >
- struct csp::type_traits::is_constraint< T >
- struct csp::Csp< VarPtr >

Namespaces

- csp

  Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.

Functions

- template<typename VarPtr >
  std::true_type **csp::type_traits::implementations::arcTest** (Arc< VarPtr >)
- std::false_type **csp::type_traits::implementations::arcTest** (...)
- template<typename VarPtr >
  std::true_type **csp::type_traits::implementations::constraintTest** (Constraint< VarPtr >)
- std::false_type **csp::type_traits::implementations::constraintTest** (...)
- template<typename VarContainer , typename ArcContainer , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(∗std::begin(std::declval< ArcContainer >()))>>::value , int >
  auto csp::make_csp (const VarContainer &variables, const ArcContainer &arcs) -> Csp< std::decay_t< decltype(std::end(arcs), std::end(variables), ∗std::begin(variables))>>
- template<typename VarContainer , typename ContraintContainer , std::enable_if_t< type_traits::is_constraint< std::remove_reference_t< decltype(∗std::begin(std::declval< ContraintContainer >()))>>::value , int >
  auto csp::make_csp (const VarContainer &variables, const ContraintContainer &constraints) -> Csp< std::decay_t< decltype(std::end(constraints), std::end(variables), ∗std::begin(variables))>>
- template<typename VarIt , typename ConstrIt , std::enable_if_t< type_traits::is_constraint< std::remove_reference_t< decltype(∗std::declval< ConstrIt >())>>::value , int >
  auto csp::make_csp (VarIt vBegin, VarIt vEnd, ConstrIt cBegin, ConstrIt cEnd) -> Csp< std::decay_t< decltype(∗++cBegin, cBegin==cEnd,++vBegin, vBegin==vEnd, ∗vBegin)>>
- template<typename VarIt , typename ArcIt , std::enable_if_t< type_traits::is_arc< std::remove_reference_t< decltype(∗std::declval< ArcIt >())>>::value , int >
  auto csp::make_csp (VarIt vBegin, VarIt vEnd, ArcIt aBegin, ArcIt aEnd) -> Csp< std::decay_t< decltype(∗++aBegin, aBegin==aEnd,++vBegin, vBegin==vEnd, ∗vBegin)>>

4.2.1 Detailed Description

This file contains the csp::Csp class which represents a complete definition of a constraint satisfaction problem. The class contains a list of Variable pointers, a list of all arcs and a map that specifies incoming arcs of each variable. The csp::Csp should be created using the provided function csp::make_csp.

Author

Tim Luchterhand

Date

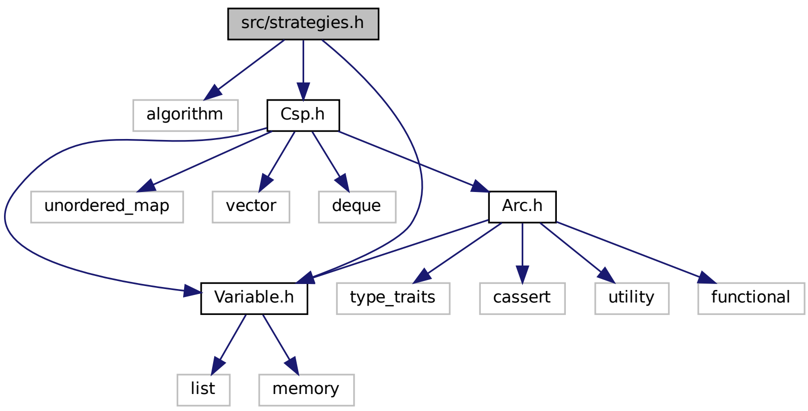
25.07.20

4.3 src/strategies.h File Reference

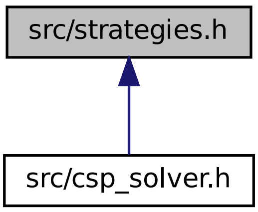
This file contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.

```
#include <algorithm>
#include "Csp.h"
#include "Variable.h"
```

Include dependency graph for strategies.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct csp::strategies::Mrv< VarPtr >
- struct csp::strategies::First< VarPtr >

Namespaces

- csp

  Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.
- csp::strategies

  contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.

4.3.1 Detailed Description

This file contains different variable choosing strategies that can be used to solve a CSP. Apart from these strategies, custom strategies can be used. A strategy must return one of the pointers to a variable stored in the given CSP.

Author

Tim Luchterhand

Date

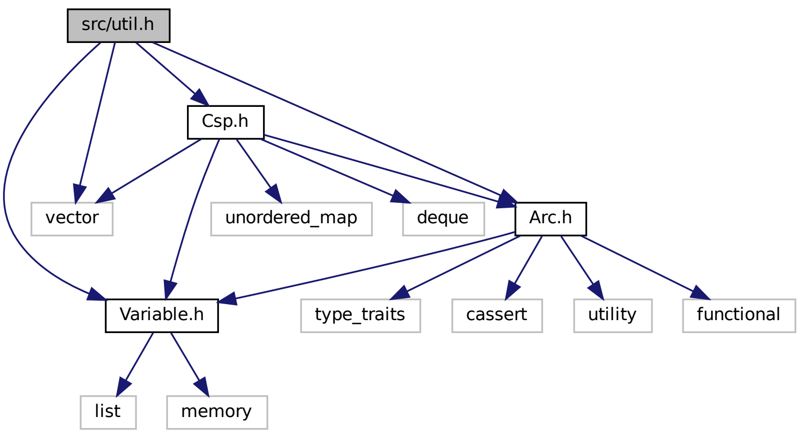
25.07.20

4.4 src/util.h File Reference

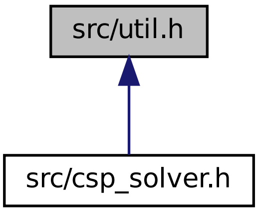
This file contains utility functions used by the search algorithm.

```
#include <vector>
#include "Variable.h"
#include "Arc.h"
#include "Csp.h"
```

Include dependency graph for util.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- [csp](#)
  Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.
- [csp::util](#)
  contains utility functions used by the search algorithm

Typedefs

- template< typename VarT >
  using **csp::util::CspCheckpoint** = std::vector< typename VarT::DomainT >

Functions

- template< typename VarPtr >
  bool csp::util::removeInconsistent (const Arc< VarPtr > &arc)
- template< typename VarPtr >
  bool csp::util::ac3 (Csp< VarPtr > &problem)
- template< typename VarPtr >
  auto csp::util::makeCspCheckpoint (const Csp< VarPtr > &problem) -> CspCheckpoint< typename Csp< VarPtr >::VarT >
- template< typename VarPtr >
  void csp::util::restoreCspFromCheckpoint (Csp< VarPtr > &problem, const CspCheckpoint< typename Csp< VarPtr >::VarT > &checkpoint)

4.4.1 Detailed Description

This file contains utility functions used by the search algorithm.

Author

Tim Luchterhand

Date

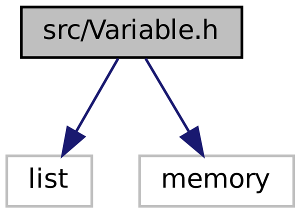
11.07.20

4.5 src/Variable.h File Reference

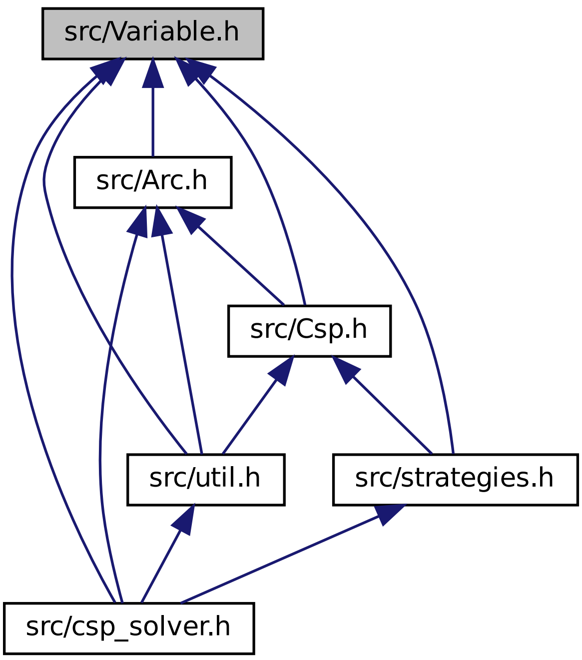
This file contains the definition of a variable which is used to describe a CSP. Custom variable types can be used by deriving from csp::Variable.

```
#include <list>
#include <memory>
```

Include dependency graph for Variable.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class csp::Variable< T, DomainType >

Namespaces

- csp

  Contains all relevant datastructures and functions for defining and solving a constraint satisfaction problem.

4.5.1 Detailed Description

This file contains the definition of a variable which is used to describe a CSP. Custom variable types can be used by deriving from csp::Variable.

Author

Tim Luchterhand

Date

11.07.20

Index