[1 Schierpinski Dreieck 2](#_Toc375343416)

[1.1 Lösungsidee 2](#_Toc375343417)

[1.2 Source 3](#_Toc375343418)

[1.2.1 SchierpinskiTriangleUnit 3](#_Toc375343419)

[1.2.2 SchierpinskiTriangleUnitTest 7](#_Toc375343420)

[1.3 Tests 8](#_Toc375343421)

[1.3.1 Fläche des Dreiecks nach n Iterationen 8](#_Toc375343422)

[1.3.2 Gesamtlänge der Seitenlängen aller platzierten Dreiecke 10](#_Toc375343423)

[2 Felderreichbarkeit 12](#_Toc375343424)

[2.1 Lösungsidee 12](#_Toc375343425)

[2.2 Source 13](#_Toc375343426)

[2.2.1 FindPathInRasterUnit 13](#_Toc375343427)

[2.2.2 FindPathInRasterUnitTest 22](#_Toc375343428)

[2.3 Tests 26](#_Toc375343429)

[2.3.1 Reset / PrintGameField 26](#_Toc375343430)

[2.3.2 SetCell 27](#_Toc375343431)

[2.3.3 PathExists 27](#_Toc375343432)

[2.3.4 LengthOfShortestPath 29](#_Toc375343433)

# Schierpinski Dreieck

## Lösungsidee

Für die Berechnung der Fläche eines Schierpinski Dreieck nach n Iterationen soll, vorerst eine Funktion implementiert werden, die die Fläche eines Dreiecks berechnen mittels folgender Formel [((a \* b) / 2)] berechnen kann. Die Berechnung der Gesamtfläche alle platzierter Dreiecke im Basisdreieck soll in einem iterativen Algorithmus, welcher in einer inneren Funktion der Hauptfunktion platziert ist, berechnet werden. Dieses Ergebnis soll anschließend in der Übergeordneten Funktion von der Fläche des Basisdreiecks subtrahiert werden.

Die Berechnung der Fläche soll wie folgt berechnet werden:

1. Summiere die Flächen aller Dreiecke, die im Basisdreieck platziert werden können.
2. Subtrahiere diese Fläche von der Fläche des Basisdreieck.

Hierbei soll folgender Ansatz umgesetzt werden:

1. Die Seitenlängen soll wie folgt berechnet werden:
   1. Länge / 2^iterationCount.
2. Iteration: 1 / 2^1 = 1 / 2 = 0,5
3. Iteration: 1 / 2^2 = 1 / 4 = 0,25
4. Iteration: 1 / 2^3 = 1 / 8 = 0,125
5. Die Anzahl der platzierten Dreiecke soll wie folgt berechnet werden:
   1. Anzahl der vorherigen Dreiecke \* 3
6. Iteration: 1 (Sonderfall bei erster Iteration)
7. Iteration: 1 \* 3 = 3
8. Iteration: 3 \* 3 = 9
9. Iteration: 9 \* 3 = 27
10. Die Fläche der Dreiecke soll wie folgt berechnet werden:
    1. Fläche eines Dreiecks \* Anzahl der platzierten Dreiecke
11. Iteration: 0,125 \* 1 = 0,125 (Sonderfall bei erster Iteration)
12. Iteration: 0,03125 \* 3 = 0,09375
13. Iteration: 0,0078125 \* 9 = 0,703125
14. Bilde die Summe über die berechneten Flächen über rekursiven Aufruf der Funktion.

Es können folgende Sonderfälle auftreten:

1. Iterationsanzahl ist kleiner gleich 0. Gib die Fläche des Basisdreiecks zurück.

(Abfrage in übergeordneter Funktion)

1. Iterationsanzahl ist gleich der aktuellen Iteration. Gib die Fläche der aktuell platzierten Dreiecke zurück. (Abbruch der Iteration)

Bei der iterativen Implementierung soll die rekursive Funktion iterativ umgesetzt werden, sodass der Ablauf derselbe ist. Als Schleife soll eine Abbruchschleife verwendet werden, um sich eine IF-Verzweigung zu ersparen, da auch auf eine Iterationsanzahl geprüft werden soll, die kleiner gleich 0 ist.

Für die Berechnung der Summe der Seitenlängen soll derselbe rekursive Ansatz verwendet werden, als bei der Berechnung der der Fläche des Dreiecks. Anstatt der Berechnung der Fläche soll hierbei die Summe der Seitenlängen der platzierten Dreiecke wie folgt berechnet werden.

Berechnete Länge \* 3 \* Anzahl der Dreiecke

Ebenso soll mit der iterativen Implementierung verfahren werden. Auch hierbei soll lediglich die Berechnung der Fläche durch die Berechnung der Seitenlänge der Dreiecke ersetzt werden.

## Source

Folgend ist der Source der Unit SchierpinskiTriangleUnit und dessen Tests angeführt.

### SchierpinskiTriangleUnit

Folgend ist der Source der Unit SchierpisnkiTriangleUnit angeführt.

{

Unit which provides utility methods for handling a Schierpinski triangle.

It supports the calculation of the area left on the base triangle for a defined

count of iterations.

Also the calculation of the sum of dimension of all placed triangles in the base triangle

for the defined iteration count is provided.

}

**UNIT** SchierpinskiTriangleUnit**;**

**INTERFACE**

{

Calculates the area of the Schierpinski triangle for the given iteration count

via a recursive implementation

@param

i: the count of iterations

@param

dimension: the length of the triangle sides

@return

the area of the triangle after the iteration count, or the area of the triangle

without iterations if i <= 0.

}

**FUNCTION** CalcAreaForItCount**(**i**,** dimension**:** INTEGER**):** REAL**;**

{

Calculates the area of the Schierpinski triangle for the given iteration count

via a iterative implementation.

@param

i: the count of iterations

@param

dimension: the length of the triangle sides

@return

the area of the triangle after the iteration count, or the area of the triangle

without iterations if i <= 0.

}

**FUNCTION** CalcAreaForItCountIt**(**i**,** dimension**:** INTEGER**):** REAL**;**

{

Calculates the sum of the dimensions of all placed triangles in a Schierpinski triangle

via a recursive implementation.

@param

i: the count of the iterations

@param

dimension: the dimension of the base triangle

@return

the sum of all dimensions of all placed triangles

}

**FUNCTION** CalcTriangleDimesnions**(**i**,** dimension**:** INTEGER**):** REAL**;**

{

Calculates the sum of the dimensions of all placed triangles in a Schierpinski triangle

via a iterative implementation.

@param

i: the count of the iterations

@param

dimension: the dimension of the base triangle

@return

the sum of all dimensions of all placed triangles

}

**FUNCTION** CalcTriangleDimesnionsIt**(**i**,** dimension**:** INTEGER**):** REAL**;**

**IMPLEMENTATION**

{ ######################## Private Function/Procedures ######################## }

{

Calculates the area of a triangle for the given dimensions.

The dimensions can be defined as decimal values.

@param

a: one side length of the triangle

@param

b: the second side of the triangle

@return

the calculated area represented by real or 0 if one/both of given dimensions

have negative values

}

**FUNCTION** CalculateTriangleArea**(**a**,** b**:** REAL**):** REAL**;**

**BEGIN**

CalculateTriangleArea **:=** 0**;**

**IF** **((**a **>** 0**)** **AND** **(**b **>** 0**))** **THEN** **BEGIN**

CalculateTriangleArea **:=** **((**a **\*** b**)** **/** 2**);**

**END;**

**END;**

{

Calculates 'value powered by n'.

@param value:

the value to powered

@param n:

the count how often the given 'value' shall be powered

@return the given 'value powered by n', where when 'n <= 0' then the neutral

element is returned, which is 1

}

**FUNCTION** Power**(**value**,** n**:** INTEGER**):** LONGINT**;**

**VAR**

i**:** INTEGER**;**

temp**:** LONGINT**;**

**BEGIN**

temp **:=** value**;**

**IF** n **>** 0 **THEN** **BEGIN**

**FOR** i **:=** 1 **TO** n **DO** **BEGIN**

temp **:=** temp **\*** value**;**

**END**

**END;**

Power **:=** temp **DIV** value**;**

**END;**

{ ######################## Public Function/Procedures ######################## }

{ Calculcate area for given iterations recursive }

**FUNCTION** CalcAreaForItCount**(**i**,** dimension**:** INTEGER**):** REAL**;**

{

Inner function which calculates the sum of the triangles which are placed

in the Shierpinski triangle at defined iterations.

@param

maxIt: the maximum iterations to handle

@param

it: the current iteration, used for the calculating the divisor for the length

of the new current placed triangles

@param

triangleCount: the count of the triangles to place on the current iteration

@param

dimension: the length of the original triangle

@return

the summary of the area of the placed triangles

}

**FUNCTION** SumPlacedTriangles**(**maxIt**,** it**:** INTEGER**;** triangleCount**:** LONGINT**;** dimension**:** REAL**):** REAL**;**

**VAR**

divisor**,** area**:** REAL**;**

**BEGIN**

{ Calculate the divisor for the dimension calculation }

divisor **:=** Power**(**2**,** it**);**

{ Calculate the area of the new placed triangles and multiply them with the new triangle count }

area **:=** CalculateTriangleArea**((**dimension **/** divisor**),** **(**dimension **/** divisor **/** 2**))** **\*** 2 **\*** triangleCount**;**

{ Break after maximum iterations are reached }

**IF** **(**it **=** maxIt**)** **THEN** **BEGIN**

SumPlacedTriangles **:=** area**;**

**END**

{ Else calculate triangle are summary}

**ELSE** **BEGIN**

SumPlacedTriangles **:=** area **+** SumPlacedTriangles**(**maxIt**,** **(**it **+** 1**),** **(**triangleCount **\*** 3**),** dimension**);**

**END;**

**END;**

**BEGIN**

{ If iterations lower than one return area of basis triangle }

**IF** **(**i **<** 1**)** **THEN** **BEGIN**

CalcAreaForItCount **:=** **(**CalculateTriangleArea**(**dimension**,** **(**dimension **/** 2**))** **\*** 2**);**

**END**

{ If iteration greater than 0 then calculate the summary of the placed triangles and subtract from basis triangle }

**ELSE** **BEGIN**

CalcAreaForItCount **:=** **(**CalculateTriangleArea**(**dimension**,** **(**dimension **/** 2**))** **\*** 2**)** **-** SumPlacedTriangles**(**i**,** 1**,** 1**,** dimension**);**

**END;**

**END;**

{ Calculate area for given iterations iterative }

**FUNCTION** CalcAreaForItCountIt**(**i**,** dimension**:** INTEGER**):** REAL**;**

**VAR**

baseArea**,** sum**,** divisor**:** REAL**;**

idx**:** INTEGER**;**

triangleCount**:** LONGINT**;**

**BEGIN**

baseArea **:=** **(**CalculateTriangleArea**(**dimension**,** **(**dimension**/**2**))** **\*** 2**);**

idx **:=** 1**;**

triangleCount **:=** 1**;**

sum **:=** 0**;**

divisor **:=** 0**;**

**WHILE** **((**i **>** 0**)** **AND** **(**idx **<=** i**))** **DO** **BEGIN**

divisor **:=** Power**(**2**,** idx**);**

sum **:=** sum **+** **(**CalculateTriangleArea**((**dimension **/** divisor**),** **(**dimension **/** divisor **/** 2**))** **\*** 2 **\*** triangleCount**);**

triangleCount **:=** triangleCount **\*** 3**;**

Inc**(**idx**);**

**END;**

CalcAreaForItCountIt **:=** baseArea **-** sum**;**

**END;**

{ Calculate the dimensions of the placed triangles for given iterations recursive }

**FUNCTION** CalcTriangleDimesnions**(**i**,** dimension**:** INTEGER**):** REAL**;**

{

Function which calculates the sum of the dimension of all placed triangles for the defined iterations.

@param

maxIt: the maximum iteration count

@param

it: the current iteration

@param

trianlgeCount: the count of the to place triangles

@param

dimension: the length of the triangles

}

**FUNCTION** SumTriangleDimensions**(**maxIt**,** it**:** INTEGER**;** triangleCount**:** LONGINT**;** dimension**:** REAL**):** REAL**;**

**VAR**

dimensionSum**,** divisor**:** REAL**;**

**BEGIN**

divisor **:=** Power**(**2**,** it**);**

dimensionSum **:=** **((**dimension **/** divisor**)** **\*** 3 **\*** triangleCount**);**

{ If iterations lower than one return 0 length }

**IF** **(**maxIt **<** 1**)** **THEN** **BEGIN**

SumTriangleDimensions **:=** 0**;**

**END**

{ Break after maximum iterations are reached }

**ELSE** **IF** **(**it **=** maxIt**)** **THEN** **BEGIN**

SumTriangleDimensions **:=** dimensionSum**;**

**END**

{ Else calculate triangle are summary}

**ELSE** **BEGIN**

SumTriangleDimensions **:=** dimensionSum **+** SumTriangleDimensions**(**maxIt**,** **(**it **+** 1**),** **(**triangleCount **\*** 3**),** dimension**);**

**END;**

**END;**

**BEGIN**

CalcTriangleDimesnions **:=** SumTriangleDimensions**(**i**,** 1**,** 1**,** dimension**);**

**END;**

{ Calculate the dimensions of the placed triangles for given iterations iterative }

**FUNCTION** CalcTriangleDimesnionsIt**(**i**,** dimension**:** INTEGER**):** REAL**;**

**VAR**

sum**,** divisor**:** REAL**;**

idx**:** INTEGER**;**

triangleCount**:** LONGINT**;**

**BEGIN**

idx **:=** 1**;**

triangleCount **:=** 1**;**

sum **:=** 0**;**

divisor **:=** 0**;**

**WHILE** **((**i **>** 0**)** **AND** **(**idx **<=** i**))** **DO** **BEGIN**

divisor **:=** Power**(**2**,** idx**);**

sum **:=** sum **+** **((**dimension **/** divisor**)** **\*** 3 **\*** triangleCount**);**

triangleCount **:=** triangleCount **\*** 3**;**

Inc**(**idx**);**

**END;**

CalcTriangleDimesnionsIt **:=** sum**;**

**END;**

**BEGIN**

**END.**

### SchierpinskiTriangleUnitTest

Folgend ist der Source der Unit SchierpinskiTriangleUnitTest angeführt.

**PROGRAM** SchierpinskiTriangleUnitTest**;**

**USES** SchierpinskiTriangleUnit**;**

**VAR**

i**:** INTEGER**;**

**BEGIN**

WriteLn**(**'########################################################'**);**

WriteLn**(**'Triangle area after n iterations left:'**);**

WriteLn**(**'########################################################'**);**

**FOR** i**:=** 0 **TO** 19 **DO** **BEGIN**

WriteLn**(**i**,** '-recursive: '**,** SchierpinskiTriangleUnit**.**CalcAreaForItCount**(**i**,** 1**):**5**);**

WriteLn**(**i**,** '-iterativ : '**,** SchierpinskiTriangleUnit**.**CalcAreaForItCountIt**(**i**,** 1**):**5**);**

**END;**

WriteLn**(**'########################################################'**);**

{ Causes range check error }

(\* WriteLn;

WriteLn('########################################################');

WriteLn('Triangle area error 20 iterations :');

WriteLn('########################################################');

WriteLn('20-recursive: ', SchierpinskiTriangleUnit.CalcAreaForItCount(20, 1):5);

WriteLn('20-iterativ : ', SchierpinskiTriangleUnit.CalcAreaForItCountIt(20, 1):5); \*)

{ Causes range check error }

(\* WriteLn;

WriteLn('########################################################');

WriteLn('Triangle area error 21 iterations :');

WriteLn('########################################################');

WriteLn('20-recursive: ', SchierpinskiTriangleUnit.CalcAreaForItCount(21, 1):5); \*)

WriteLn**;**

WriteLn**(**'########################################################'**);**

WriteLn**(**'Placed triangle dimensions after n iterations:'**);**

WriteLn**(**'########################################################'**);**

**FOR** i**:=** 0 **TO** 19 **DO** **BEGIN**

WriteLn**(**i**,** '-recursive: '**,** SchierpinskiTriangleUnit**.**CalcTriangleDimesnions**(**i**,** 1**):**5**);**

WriteLn**(**i**,** '-iterativ : '**,** SchierpinskiTriangleUnit**.**CalcTriangleDimesnionsIt**(**i**,** 1**):**5**);**

**END;**

WriteLn**(**'########################################################'**);**

{ Causes range check error }

(\* WriteLn;

WriteLn('########################################################');

WriteLn('Placed triangle dimensions after 20 iterations:');

WriteLn('########################################################');

WriteLn('20-recursive: ', SchierpinskiTriangleUnit.CalcTriangleDimesnions(20, 1):5);

WriteLn('20-iterativ : ', SchierpinskiTriangleUnit.CalcTriangleDimesnionsIt(20, 1):5); \*)

{ Causes range check error }

(\* WriteLn;

WriteLn('########################################################');

WriteLn('Placed triangle dimensions after 21 iterations:');

WriteLn('########################################################');

WriteLn('20-recursive: ', SchierpinskiTriangleUnit.CalcTriangleDimesnions(21, 1):5); \*)

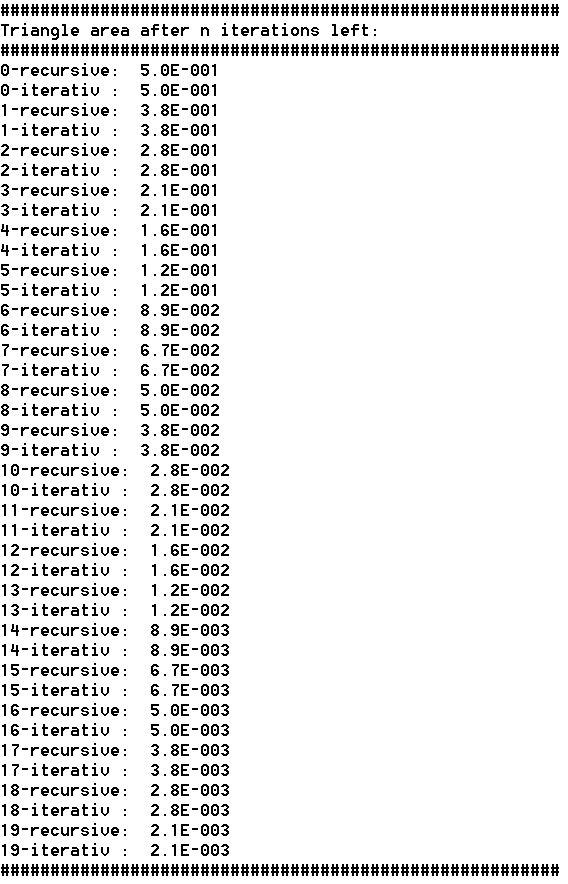
**END.**

## Tests

Folgend sind die Tests der SchierpinskiUnit angeführt.

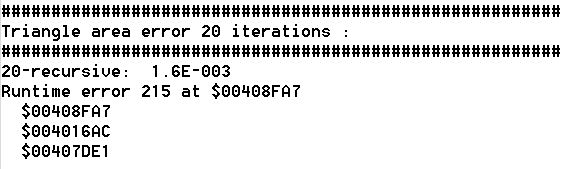
### Fläche des Dreiecks nach n Iterationen

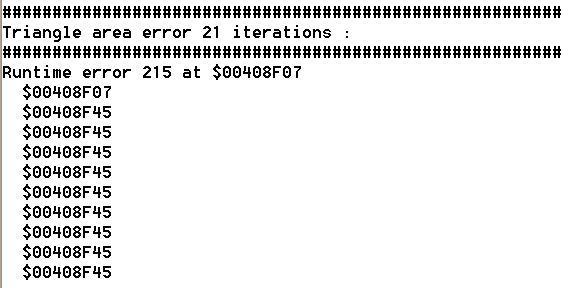
Dieser Test testet die Prozedur CalcAreaForItCount (rekursiv) und CalcAreaForItCountIt (iterativ) von 0 bis 21 Iterationen.



Bei 0 Iterationen wird die Fläche des Schierpinski Dreiecks zurückgeliefert, da hierbei keine Dreiecke platziert werden können.

Ansonsten werden die Flächen der platzierten Dreiecke im Schierpinski Dreiecks von der Fläche des Schierpinski Dreiecks abgezogen.

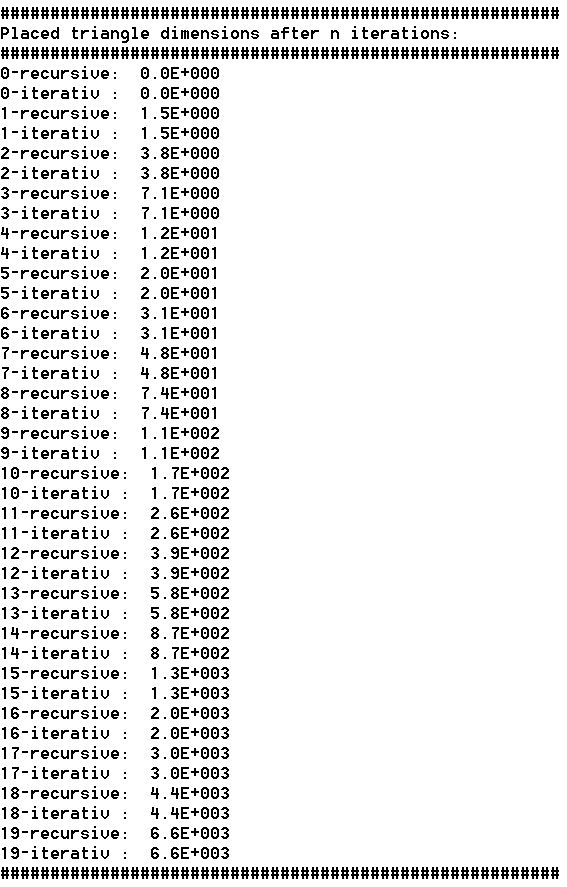




Bei 20 Iterationen zeigen sich die Schwächen der Iterativen Implementierung. Da hier in einer Schleife die Berechnungen erfolgen tritt dieser Fehler bereits bei der 20 Iteration auf und nicht bei der 21, so wie bei der rekursiven Implementierung. Dies liegt daran, dass bei der rekursiven Implementierung die Berechnung der Anzahl für die nächste Iteration wegfällt, bei der Iterativen jedoch in der Schleife noch erfolgt. Dieser Fehler rührt daher, dass die Anzahl der platzierten Dreiecke so stark zunimmt, sodass der Wertebereich des LONGINT Datentyp in Pascal nicht mehr ausreicht. Und da beim Compiler der Range Check aktiviert ist, wird dieser Fehler geworfen wenn die Anzahl der Dreiecke zu groß für den verwendeten Datentyp ist. Ohne Range Check würde hier ein Überlauf stattfinden.

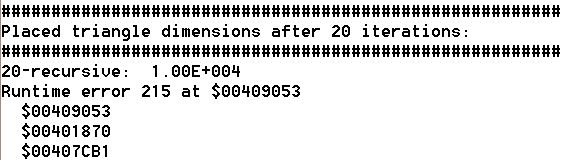
### Gesamtlänge der Seitenlängen aller platzierten Dreiecke

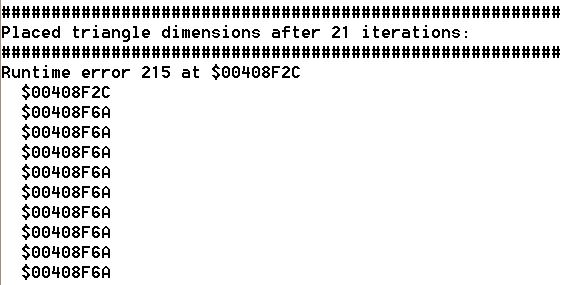
Dieser Test testet die Prozedur CalcTriangleDimesnions (recursiv) und CalcTriangleDimesnionsIt (iterativ) von 0 bis 21 Iterationen.



Bei 0 Iterationen wird als Resultat 0 zurückgeliefert, da hierbei keine Dreiecke im Schierpinski Dreieck platziert werden konnten.

Ansonsten werden die Seitenlängen aller platzierten Dreiecke, die im Schierpinski Dreieck platziert wurden, aufsummiert und zurückgeliefert.





Da die beiden Algorithmen sich sehr ähneln und die Berechnung der Anzahl der platzierten Dreiecke gleich ist, tritt auch hier der Fehler des Range Check auf, da auch hier die Anzahl der platzierten Dreiecke zu groß für den verwendeten Datentyp wird. Der Grund ist derselbe, wie im vorherigen Test beschrieben.

# Felderreichbarkeit

## Lösungsidee

Als für das Spielfeld soll ein Datentyp spezifiziert werden, welches ein zwei dimensionales Array beinhaltet, welches wiederum die einzelnen Felder beinhaltet (x – y Koordinate des Spielfelds).

Für die einzelnen Felder des Spielfelds soll ebenfalls ein Datentyp spezifiziert werden, welches folgende Attribute spezifiziert:

1. X:

Die X Koordinate des Feldes

1. Y:

Die Y Koordinate des Feldes

1. Wall:

TRUE wenn dieses Feld eine Wand darstellt.

1. Symbol:

Das Symbol des Feldes (Könnte vom Spieler definiert werden)

Das einzelne Feld soll seine Koordinaten beinhalten, damit in etwaigen Funktionen oder Prozeduren nicht die Koordinaten mitübergeben werden müssen. Des Weiteren wird hiermit eine Fehlerquelle ausgeschlossen, da hier explizit die Koordinaten des Feldes geändert werden müssten um Positionsfehler hervorzurufen.

Um Überläufe zu vermeiden soll für die Spielfeldgröße bezüglich der x und y Achse ein eigener Datentyp spezifiziert werden, welcher den Wertebereich spezifiziert, der für die beiden Koordinaten zur Verfügung stehen darf.

Für das Suchen der Startposition soll ein eigener Datentyp spezifiziert werden, der das Feld und einen Zähler hält, welcher dafür gedacht ist um herauszufinden, wie weit das Feld vom Ziel entfernt ist.

Dieser soll als Element in eine zweifach verketteten zyklischen Liste mit Ankerelement gespeichert werden können. Diese Liste ist für den Algorithmus unerlässlich.

Das Spielfeld soll innerhalb der Unit, nicht sichtbar nach außen, gehalten werden. Über eine Reset Prozedur soll das Spielfeld wieder zurückgesetzt werden, wobei die alle Felder innerhalb des Spielfeldes leere Felder seien sollen.

Suchalgorithmus:

1. Prüfe ob entweder der Start oder Ziel eine Wand ist. Wenn ja führe den Hauptalgorithmus nicht aus, ansonsten führe in aus.
2. Erstelle eine Liste und füge als erstes Element Start/Ziel hinzu mit einem Zähler von 0.

(x, y, 0)

1. Baue eine temporäre Liste auf und füge alle Nachbarn hinzu mit Zähler + 1.

(x+1, y, 1), (x-1, y, 1), (x, y-1, 1), (x, y+1, 1)

1. Entferne alle Elemente aus der temporären Liste, die in der Hauptliste vorhanden sind und entweder eine Wand sind, oder dieselben Koordinaten haben, wobei der Zähler größer gleich dem Zähler des Elements der temporären Liste sein muss.
2. Füge die übrig gebliebenen Elemente in der Hauptliste hinzu.
3. Führe diesen Algorithmus ab Schritt 3 für alle verbliebenen Elemente der Hauptliste aus, solange keine Elemente mehr übrig sind oder ein Element der temporären Liste, die Koordinaten von Ziel/Start hat. Ist dies nicht der Fall, so kann die Position nicht erreicht werden.

Der beschriebe Algorithmus soll für die Prozeduren PathExists (rekursiv), PathExistsIt (iterativ), LengthOfShortestPath (rekursiv) verwendet werden, wobei die Hauptfunktionalitäten von Schritt 3 – 5 in eine eigene Prozedur ausgelagert werden sollen, da hier der meiste Implementierungsaufwand besteht. Um die implementierten Algorithmen übersichtlich zu halten, sollen Hilfsprozeduren und –Funktionen implementiert werden, die die Handhabung mit den Feldern und verwendeten Listen erleichtern sollen. Ebenso sollen damit die einzelnen Funktionalitäten getrennt und gekapselt werden.

Die Prozedur PrintGameField soll das erstellte Spielfeld auf der Konsole ausgeben.

Die Prozedur SetCell soll auf der gegebenen Position ein Feld setzen, das entweder eine Wand oder ein freies Feld ist, je nachdem wie der Aufrufer dies definiert.

## Source

Folgend ist der Source der FindPathInRasterUnit und FindPathInRasterunitTest angeführt.

### FindPathInRasterUnit

Folgend ist der Source der FindPathInRasterUnit angeführt.

{

Unit which provides the functionality to find a field from a start field within a raster.

It can also determine the shortest path to the intended field.

}

**UNIT** FindPathInRaster**;**

**INTERFACE**

**TYPE**

{ The range of the x coordinate }

xRange **=** 1**..**20**;**

{ The range of the y coordinate }

yRange **=** 1**..**20**;**

{

Resets the game by creating a new game field with no walls.

}

**PROCEDURE** Reset**;**

{

Prints the game field to the console.

}

**PROCEDURE** PrintGameField**;**

{

Sets a raster FieldCell as a wall or an empty FieldCell.

The existing type of the raster FieldCell will be overwritten.

@param

x: the x coordinate of the FieldCell

@param

y: the y coordinate of the FieldCell

@param

wall: true if the FieldCell shall be set as a wall, false otherwise

}

**PROCEDURE** SetCell**(**x**:** xRange**;** y**:** yRange**;** wall**:** BOOLEAN**);**

{

Answers the question if the given FieldCell can be reached of the given position

via a recursive implementation.

@param

ax, ay: the coordinates of the start position

@param

bx, by: the goal position

@return

true if the position can reached with the given steps, false otherwise

}

**FUNCTION** PathExists**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** BOOLEAN**;**

{

Answers the question if the given FieldCell can be reached of the given position

via a iterative implementation.

@param

ax, ay: the coordinates of the start position

@param

bx, by: the goal position

@return

true if the position can reached with the given steps, false otherwise

}

**FUNCTION** PathExistsIt**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** BOOLEAN**;**

{

This function gets the shortest path length.

@param

x: the x coordinate of the start field

@param

y: the y coordinate of the start field

@param

x: the x coordinate of the goal field

@param

y: the y coordinate of the goal field

@return

the length of the shortest path or -1 if no path could be found

}

**FUNCTION** ShortestPathLength**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** INTEGER**;**

**IMPLEMENTATION**

**TYPE**

{ The type of the FieldCell }

FieldCell **=** **RECORD**

wall**:** BOOLEAN**;**

symbol**:** **STRING;**

y**:** yRange**;**

x**:** xRange**;**

**END;**

{ The type for the game field which holds the cells }

GameField **=** **RECORD**

raster**:** **ARRAY[**yRange**,** xRange**]** **OF** FieldCell**;**

**END;**

{ AType used for the found cells in the raster }

FieldCellNode **=** **^**FoundFieldCell**;**

FoundFieldCell **=** **RECORD**

cell**:** FieldCell**;**

count**:** INTEGER**;**

prev**,** next**:** FieldCellNode**;**

**END;**

{ The type for the list of FieldCellnode }

FoundFieldList **=** FieldCellNode**;**

**VAR**

{ The GameField instance visible only for this module }

game**:** GameField**;**

{ ###################### Private Functions and Procedures ###################### }

{

Creates a raster field of the specified type.

@param

x: the x coordinate where the field resides

@param

y: the y coordinate where the field resides

@param

wall: if true then the field will be created as a wall, as an empty field otherwise.

@param

symbol: the symbol used for the FieldCell

@return

the created FieldCell instance

}

**FUNCTION** CreateRasterField**(**x**:** xRange**;** y**:** yRange**;** wall**:** BOOLEAN**;** symbol**:** **STRING):** FieldCell**;**

**VAR**

cell**:** FieldCell**;**

**BEGIN**

cell**.**x **:=** x**;**

cell**.**y **:=** y**;**

cell**.**wall **:=** wall**;**

cell**.**symbol **:=** symbol**;**

CreateRasterField **:=** cell**;**

**END;**

{

Answers the question if the given coordinates are valid ones, by checking the ranges defined by the

custom type.

@param

x: the x coordinate

@param

y: the y coordinate

@return

true if the given coordinates are valid

}

**FUNCTION** IsValidCoordinate**(**x**,** y**:** INTEGER**):** BOOLEAN**;**

**BEGIN**

IsValidCoordinate **:=** **((**x **>=** Low**(**xRange**))** **AND** **(**x **<=** High**(**xRange**))** **AND** **(**y **>=** Low**(**yRange**))** **AND** **(**y **<=** High**(**yRange**)));**

**END;**

{

Answers the question if the two given fields are the same by comparison of their coordinates.

@param

f1: the first FieldCell instance

@param

f2: the second FieldCell instance

@return

true if the both fields are equal, false otherwise

}

**FUNCTION** IsSameField**(**f1**,** f2**:** FieldCell**):** BOOLEAN**;**

**BEGIN**

IsSameField **:=** **((**f1**.**x **=** f2**.**x**)** **AND** **(**f1**.**y **=** f2**.**y**));**

**END;**

{ ######################### List helper function and procedures ######################### }

{

Creates a FieldCellNode instance.

@param

cell: the FieldCell to be hold by the created FieldCellNode instance

@param

count: the count to be hold by the FieldCellNode instance

@return

the created FieldCellNode instance

}

**FUNCTION** CreateFoundFieldCellNode**(**cell**:** FieldCell**;** count**:** INTEGER**):** FieldCellNode**;**

**VAR**

node**:** FieldCellNode**;**

**BEGIN**

New**(**node**);**

node**^.**cell **:=** cell**;**

node**^.**count **:=** count**;**

node**^.**next **:=** node**;**

node**^.**prev **:=** node**;**

CreateFoundFieldCellNode **:=** node**;**

**END;**

{

Appends a FieldCellNode to the end of the given list.

@param

list: the list to append the node to the end

@param

node: the node to be added to the end of the list

}

**PROCEDURE** AppendFoundFieldNode**(**list**:** FoundFieldList**;** node**:** FieldCellNode**);**

**BEGIN**

node**^.**next **:=** list**;**

node**^.**prev **:=** list**^.**prev**;**

list**^.**prev**^.**next **:=** node**;**

list**^.**prev **:=** node**;**

**END;**

{

Destroys the given list.

@param

list: the list to be destroyed

@return

list: the destroyed list which will be NIL

}

**PROCEDURE** DestroyFoundCellList**(VAR** list**:** FoundFieldList**);**

**VAR**

node**,** next**:** FieldCellNode**;**

**BEGIN**

node **:=** list**^.**next**;**

**WHILE** **(**node **<>** list**)** **DO** **BEGIN**

next **:=** node**^.**next**;**

Dispose**(**node**);**

node **:=** next**;**

**END;**

Dispose**(**node**);**

list **:=** **NIL;**

**END;**

{

Appends the neighbours of the FieldCell of the given coordinates to the given list.

@param

x: the x axis coordinate

@param

y: the y axis coordinate

@param

count: the count for FieldCellNode instance

@param

list: the list to add the neighbours to the list

}

**PROCEDURE** AppendNeighbourFieldCells**(**x**,** y**,** count**:** INTEGER**;** list**:** FoundFieldList**);**

**BEGIN**

{ Append right neighbour cell if possible }

**IF** **(**IsValidCoordinate**((**x **+** 1**),** y**))** **THEN** **BEGIN**

AppendFoundFieldNode**(**list**,** CreateFoundFieldCellNode**(**game**.**raster**[**y**][**x **+** 1**],** count**));**

**END;**

{ Append left neighbour cell if possible }

**IF** **(**IsValidCoordinate**(**x **-** 1**,** y**))** **THEN** **BEGIN**

AppendFoundFieldNode**(**list**,** CreateFoundFieldCellNode**(**game**.**raster**[**y**][**x **-** 1**],** count**));**

**END;**

{ Append bottom neighbour cell if possible }

**IF** **(**IsValidCoordinate**(**x**,** y **+** 1**))** **THEN** **BEGIN**

AppendFoundFieldNode**(**list**,** CreateFoundFieldCellNode**(**game**.**raster**[**y **+** 1**][**x**],** count**));**

**END;**

{ Append top neighbour cell if possible }

**IF** **(**IsValidCoordinate**(**x**,** y **-** 1**))** **THEN** **BEGIN**

AppendFoundFieldNode**(**list**,** CreateFoundFieldCellNode**(**game**.**raster**[**y **-** 1**][**x**],** count**));**

**END;**

**END;**

{

Deletes a FieldCellNode from the given list.

@param

list: the list to remove the element from

@param

node: the node to remove from the list

}

**PROCEDURE** DeleteFoundFieldNode**(**list**:** FoundFieldList**;** removeNode**:** FieldCellNode**);**

**VAR**

node**:** FieldCellNode**;**

**BEGIN**

node **:=** list**^.**next**;**

{ Search for the cell }

**WHILE** **((NOT** IsSameField**(**node**^.**cell**,** removeNode**^.**cell**))** **AND** **(**node **<>** list**))** **DO** **BEGIN**

node **:=** node**^.**next**;**

**END;**

{ If cell has been found }

**IF** **(**node **<>** list**)** **THEN** **BEGIN**

node**^.**prev**^.**next **:=** node**^.**next**;**

node**^.**next**^.**prev **:=** node**^.**prev**;**

node**^.**next **:=** **NIL;**

node**^.**prev **:=** **NIL;**

Dispose**(**node**);**

**END;**

**END;**

{

Gets the FieldCellNode instance for the given FieldCell.

@param

list: the list to search for the FieldCell

@param

cell: the FieldCell to search for

@return

the FoudnFieldNode instance, NIL if the FieldCell could not be found in the list

}

**FUNCTION** GetFoundFieldNode**(**list**:** FoundFieldList**;** cell**:** FieldCell**):** FieldCellNode**;**

**VAR**

node**:** FieldCellNode**;**

**BEGIN**

GetFoundFieldNode **:=** **NIL;**

node **:=** list**^.**next**;**

**WHILE** **((NOT** IsSameField**(**node**^.**cell**,** cell**))** **AND** **(**node **<>** list**))** **DO** **BEGIN**

node **:=** node**^.**next**;**

**END;**

**IF** **(**node **<>** list**)** **THEN** **BEGIN**

(\* WriteLn('found node: ', node^.cell.x, ',', node^.cell.y); \*)

GetFoundFieldNode **:=** node**;**

**END;**

**END;**

{

Cleans the temporary list by removing all elements of the target list which are present in the list and which fit the following condition.

1. List contains element with the same coordinate and with a equal or lower count as the target element

2. Target element is a wall

@param

target: the target to remove the elements of

@param

list: the list to compare the elements of

}

**PROCEDURE** CleanFoundCellList**(**target**,** list**:** FoundFieldList**);**

**VAR**

node**,** next**,** temp**:** FieldCellNode**;**

**BEGIN**

node **:=** target**^.**next**;**

**WHILE** **(**node **<>** target**)** **DO** **BEGIN**

next **:=** node**^.**next**;**

{ Delete element if it is a wall }

**IF** **(**node**^.**cell**.**wall**)** **THEN** **BEGIN**

DeleteFoundFieldNode**(**target**,** node**);**

**END**

{ Else check for same coordinate and equal or higher value of target element compared to found list element}

**ELSE** **BEGIN**

temp **:=** GetFoundFieldNode**(**list**,** node**^.**cell**);**

**IF** **((**temp **<>** **NIL)** **AND** **(**node**^.**count **>=** temp**^.**count**))** **THEN** **BEGIN**

DeleteFoundFieldNode**(**target**,** node**);**

**END;**

**END;**

node **:=** next**;**

**END;**

**END;**

{

Adds all elements of the source list to the end of the target list.

@param

target: the list to add elements on the end

@param

source: the list to add its element to the target list

}

**PROCEDURE** AddAllCellsToList**(**target**,** source**:** FoundFieldList**);**

**VAR**

node**:** FieldCellNode**;**

**BEGIN**

node **:=** source**^.**next**;**

**WHILE** **(**node **<>** source**)** **DO** **BEGIN**

AppendFoundFieldNode**(**target**,** CreateFoundFieldCellNode**(**node**^.**cell**,** node**^.**count**));**

node **:=** node**^.**next**;**

**END;**

**END;**

{

Answers the question if the given lists contains the given FieldCell.

@param

list: the list to search for the FieldCell

@param

cell: the FieldCell to search on the list elements

@return

true if the given list contains the FieldCell, false otherwise

}

**FUNCTION** ContainsField**(**list**:** FoundFieldList**;** cell**:** FieldCell**):** BOOLEAN**;**

**BEGIN**

ContainsField **:=** **(**GetFoundFieldNode**(**list**,** cell**)** **<>** **NIL);**

**END;**

{

Implements the main function of the algorithm which is used to find the path to the position in the game field.

@param

node the node to work with

@param

visited: the FoundFieldList which gets elements added, if possible

@param

count the current count to set on the found fields

@return

visited: the FoundFieldList which maybe got modified

}

**PROCEDURE** HandleCurrentFoundCell**(**node**:** FieldCellNode**;** **VAR** visited**:** FoundFieldList**);**

**VAR**

tempList**:** FoundFieldList**;**

**BEGIN**

{ WriteLn; }

{ WriteLn('current-node: ', node^.cell.x, ',', node^.cell.y); }

{ Create the temporary list }

tempList **:=** CreateFoundFieldCellNode**(**CreateRasterField**(**Low**(**xRange**),** Low**(**yRange**),** false**,** ''**),** **-**1**);**

{ WriteLn('created found node'); }

{ Add the neighbours to the temporary list }

AppendNeighbourFieldCells**(**node**^.**cell**.**x**,** node**^.**cell**.**y**,** **(**node**^.**count **+** 1**),** tempList**);**

{ WriteLn('appended neighbours'); }

{ Clean the temporary list }

CleanFoundCellList**(**tempList**,** visited**);**

{ WriteLn('cleaned temp'); }

{ Append remaining items to the visited list }

AddAllCellsToList**(**visited**,** tempList**);**

{ WriteLn('added cells to visited'); }

{ Destroy the temporary list }

DestroyFoundCellList**(**tempList**);**

{ WriteLn('destroyed temp'); }

**END;**

{

Gets the shortest path by comparison of the neighbour fields if they have a lower count value.

@param

foundCell: the found FieldCellNode instance

@param

list: The list which has to contain the calculated neighbours

@return

the lowest count value of the path

}

**FUNCTION** GetShortestCountOfNeighbours**(**foundCell**:** FieldCellNode**;** list**:** FoundFieldList**):** INTEGER**;**

**VAR**

x**,** y**,** count**:** INTEGER**;**

found**:** FieldCellNode**;**

cell**:** FieldCell**;**

**BEGIN**

x **:=** 0**;**

y **:=** 0**;**

count **:=** foundCell**^.**count**;**

{ Get count value of right neighbour }

x **:=** foundCell**^.**cell**.**x **+** 1**;**

y **:=** foundCell**^.**cell**.**y**;**

**IF** **(**IsValidCoordinate**(**x**,** y**))** **THEN** **BEGIN**

cell **:=** CreateRasterField**(**x**,** y**,** false**,** ''**);**

found **:=** GetFoundFieldNode**(**list**,** cell**);**

**IF** **((**found **<>** **NIL)** **AND** **(**found**^.**count **<** count**))** **THEN** **BEGIN**

count **:=** found**^.**count**;**

**END;**

**END;**

{ Get count value of left neighbour }

x **:=** foundCell**^.**cell**.**x **-** 1**;**

y **:=** foundCell**^.**cell**.**y**;**

**IF** **(**IsValidCoordinate**(**x**,** y**))** **THEN** **BEGIN**

cell **:=** CreateRasterField**(**x**,** y**,** false**,** ''**);**

found **:=** GetFoundFieldNode**(**list**,** cell**);**

**IF** **((**found **<>** **NIL)** **AND** **(**found**^.**count **<** count**))** **THEN** **BEGIN**

count **:=** found**^.**count**;**

**END;**

**END;**

{ Get count value of top neighbour }

x **:=** foundCell**^.**cell**.**x**;**

y **:=** foundCell**^.**cell**.**y **+** 1**;**

**IF** **(**IsValidCoordinate**(**x**,** y**))** **THEN** **BEGIN**

cell **:=** CreateRasterField**(**x**,** y**,** false**,** ''**);**

found **:=** GetFoundFieldNode**(**list**,** cell**);**

**IF** **((**found **<>** **NIL)** **AND** **(**found**^.**count **<** count**))** **THEN** **BEGIN**

count **:=** found**^.**count**;**

**END;**

**END;**

{ Get count value of top neighbour }

x **:=** foundCell**^.**cell**.**x**;**

y **:=** foundCell**^.**cell**.**y **-** 1**;**

**IF** **(**IsValidCoordinate**(**x**,** y**))** **THEN** **BEGIN**

cell **:=** CreateRasterField**(**x**,** y**,** false**,** ''**);**

found **:=** GetFoundFieldNode**(**list**,** cell**);**

**IF** **((**found **<>** **NIL)** **AND** **(**found**^.**count **<** count**))** **THEN** **BEGIN**

count **:=** found**^.**count**;**

**END;**

**END;**

GetShortestCountOfNeighbours **:=** count**;**

**END;**

{ ###################### Public Functions and Procedures ###################### }

{ Reset the memory }

**PROCEDURE** Reset**;**

**VAR**

i**,** j**:** INTEGER**;**

**BEGIN**

**FOR** i **:=** Low**(**yRange**)** **TO** High**(**yRange**)** **DO** **BEGIN**

**FOR** j **:=** Low**(**xRange**)** **TO** High**(**xRange**)** **DO** **BEGIN**

game**.**raster**[**i**][**j**]** **:=** CreateRasterField**(**j**,** i**,** false**,** '.'**);**

**END;**

**END;**

**END;**

{ Print game field }

**PROCEDURE** PrintGameField**;**

**VAR**

i**,** j**:** INTEGER**;**

**BEGIN**

**FOR** i **:=** Low**(**game**.**raster**)** **TO** High**(**game**.**raster**)** **DO** **BEGIN**

**FOR** j **:=** Low**(**game**.**raster**[**i**])** **TO** High**(**game**.**raster**[**i**])** **DO** **BEGIN**

Write**(**game**.**raster**[**i**][**j**].**symbol**:**3**);**

**END;**

WriteLn**;**

**END;**

**END;**

{ Sets a raster field as a wall or empty }

**PROCEDURE** SetCell**(**x**:** xRange**;** y**:** yRange**;** wall**:** BOOLEAN**);**

**VAR**

s**:** **STRING;**

**BEGIN**

**IF** **(**wall**)** **THEN** **BEGIN**

s **:=** '#'

**END**

**ELSE** **BEGIN**

s **:=** '.'**;**

**END;**

game**.**raster**[**y**][**x**]** **:=** CreateRasterField**(**x**,** y**,** wall**,** s**);**

**END;**

{ TODO: Checks for path exists }

**FUNCTION** PathExists**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** BOOLEAN**;**

**FUNCTION** Exists**(**goal**:** FieldCell**;** **VAR** node**:** FieldCellNode**;** **VAR** visited**:** FoundFieldList**):** BOOLEAN**;**

**BEGIN**

{ If last element has been reached }

**IF** **(**node **=** visited**)** **THEN** **BEGIN**

Exists **:=** false**;**

(\* WriteLn('could not find path'); \*)

**END**

{ If start is part of the visited list after first iteration }

**ELSE** **IF** **(**IsSameField**(**goal**,** node**^.**cell**))** **THEN** **BEGIN**

Exists **:=** true**;**

(\* WriteLn('found node recursive: ', node^.cell.x, ',', node^.cell.y); \*)

**END**

{ Search for start position }

**ELSE** **BEGIN**

HandleCurrentFoundCell**(**node**,** visited**);**

(\* WriteLn('new count: ', count); \*)

Exists **:=** Exists**(**goal**,** node**^.**next**,** visited**);**

**END;**

**END;**

**VAR**

visited**:** FoundFieldList**;**

goal**,** start**:** FieldCell**;**

**BEGIN**

start **:=** game**.**raster**[**ay**][**ax**];**

goal **:=** game**.**raster**[**by**][**bx**];**

{ Set symbol to see the start and goal on the game field }

start**.**symbol **:=** 'S'**;**

game**.**raster**[**ay**][**ax**]** **:=** start**;**

goal**.**symbol **:=** 'G'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**IF** **(**start**.**wall**)** **THEN** **BEGIN**

start**.**symbol **:=** 'SW'**;**

game**.**raster**[**ay**][**ax**]** **:=** start**;**

**END**

**ELSE** **IF** **(**goal**.**wall**)** **THEN** **BEGIN**

goal**.**symbol **:=** 'GW'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**END;**

{ If one of the given cells is a wall break here }

**IF** **((**start**.**wall**)** **OR** **(**goal**.**wall**)** **OR** **(**IsSameField**(**goal**,** start**)))** **THEN** **BEGIN**

PathExists **:=** false**;**

**END**

{ Else try to find out the path to the position }

**ELSE** **BEGIN**

visited **:=** CreateFoundFieldCellNode**(**goal**,** **-**1**);**

AppendFoundFieldNode**(**visited**,** CreateFoundFieldCellNode**(**start**,** 0**));**

PathExists **:=** Exists**(**goal**,** visited**^.**next**,** visited**);**

DestroyFoundCellList**(**visited**);**

**END;**

**END;**

{ Checks for path exists }

**FUNCTION** PathExistsIt**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** BOOLEAN**;**

**VAR**

visited**:** FoundFieldList**;**

node**:** FieldCellNode**;**

goal**,** start**:** FieldCell**;**

**BEGIN**

start **:=** game**.**raster**[**ay**][**ax**];**

goal **:=** game**.**raster**[**by**][**bx**];**

{ Set symbol to see the start and goal on the game field }

start**.**symbol **:=** 'S'**;**

game**.**raster**[**ay**][**ax**]** **:=** start**;**

goal**.**symbol **:=** 'G'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**IF** **(**start**.**wall**)** **THEN** **BEGIN**

start**.**symbol **:=** 'SW'**;**

game**.**raster**[**ay**][**ax**]** **:=** start**;**

**END**

**ELSE** **IF** **(**goal**.**wall**)** **THEN** **BEGIN**

goal**.**symbol **:=** 'GW'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**END;**

{ If one of the given cells is a wall break here }

**IF** **((**start**.**wall**)** **OR** **(**goal**.**wall**)** **OR** **(**IsSameField**(**goal**,** start**)))** **THEN** **BEGIN**

PathExistsIt **:=** false**;**

**END**

{ Else try to find out the path to the position }

**ELSE** **BEGIN**

visited **:=** CreateFoundFieldCellNode**(**goal**,** **-**1**);**

AppendFoundFieldNode**(**visited**,** CreateFoundFieldCellNode**(**start**,** 0**));**

node **:=** visited**^.**next**;**

**WHILE** **((**node **<>** visited**)** **AND** **(NOT** IsSameField**(**node**^.**cell**,** goal**)))** **DO** **BEGIN**

HandleCurrentFoundCell**(**node**,** visited**);**

node **:=** node**^.**next**;**

**END;**

PathExistsIt **:=** **(**node **<>** visited**);**

(\* IF (node <> visited) THEN BEGIN

WriteLn('found node iterative: ',node^.cell.x, ',', node^.cell.y);

END; \*)

DestroyFoundCellList**(**visited**);**

**END;**

**END;**

{ Gets the shortest path length }

**FUNCTION** ShortestPathLength**(**ax**:** xRange**;** ay**:** yRange**;** bx**:** xRange**;** by**:** yRange**):** INTEGER**;**

**VAR**

visited**:** FoundFieldList**;**

node**:** FieldCellNode**;**

goal**,** start**:** FieldCell**;**

s**:** **STRING;**

**BEGIN**

ShortestPathLength **:=** **-**1**;**

start **:=** game**.**raster**[**ay**][**ax**];**

goal **:=** game**.**raster**[**by**][**bx**];**

{ Set symbol to see the start and goal on the game field }

goal**.**symbol **:=** 'G'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**IF** **(**goal**.**wall**)** **THEN** **BEGIN**

goal**.**symbol **:=** 'GW'**;**

game**.**raster**[**by**][**bx**]** **:=** goal**;**

**END;**

{ If one of the given cells is a wall break here }

**IF** **((NOT** start**.**wall**)** **AND** **(NOT** goal**.**wall**)** **AND** **(NOT** IsSameField**(**goal**,** start**)))** **THEN** **BEGIN**

visited **:=** CreateFoundFieldCellNode**(**goal**,** **-**1**);**

AppendFoundFieldNode**(**visited**,** CreateFoundFieldCellNode**(**start**,** 0**));**

node **:=** visited**^.**next**;**

**WHILE** **((**node **<>** visited**)** **AND** **(NOT** IsSameField**(**node**^.**cell**,** goal**)))** **DO** **BEGIN**

{ WriteLn('x: ', node^.cell.x, ' y: ', node^.cell.y, ' count: ', node^.count); }

Str**(**node**^.**count**,** s**);**

game**.**raster**[**node**^.**cell**.**y**][**node**^.**cell**.**x**]** **:=** CreateRasterField**(**node**^.**cell**.**x**,** node**^.**cell**.**y**,** false**,** s**);**

HandleCurrentFoundCell**(**node**,** visited**);**

node **:=** node**^.**next**;**

**END;**

**IF** **(**node **<>** visited**)** **THEN** **BEGIN**

{ WriteLn('x: ', node^.cell.x, ' y: ', node^.cell.y, ' count: ', node^.count); }

ShortestPathLength **:=** GetShortestCountOfNeighbours**(**node**,** visited**)** **+** 1**;**

**END;**

DestroyFoundCellList**(**visited**);**

**END;**

**END;**

**BEGIN**

{ Creates a empty game field }

Reset**;**

**END.**

### FindPathInRasterUnitTest

Folgend ist der Source der FindPathInRasterUnitTest angeführt.

**PROGRAM** FindPathInRasterUnitTest**;**

**USES** FindPathInRaster**;**

{ Tests the reset procedure }

**PROCEDURE** TestReset**;**

**BEGIN**

WriteLn**;**

FindPathInRaster**.**Reset**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'Before reset'**);**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**SetCell**(**1**,**1**,**true**);**

FindPathInRaster**.**SetCell**(**10**,**16**,**true**);**

FindPathInRaster**.**SetCell**(**11**,**11**,**true**);**

FindPathInRaster**.**SetCell**(**13**,**15**,**true**);**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

FindPathInRaster**.**Reset**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'After reset'**);**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**PrintGameField**;**

FindPathInRaster**.**Reset**;**

**END;**

{ Tests the SetCell procedure }

**PROCEDURE** TestSetCell**;**

**BEGIN**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'Set field on raster'**);**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**Low**(**xRange**),** Low**(**yRange**),** true**);**

FindPathInRaster**.**SetCell**(**Low**(**xRange**),** High**(**yRange**),** true**);**

FindPathInRaster**.**SetCell**(**High**(**xRange**),** Low**(**yRange**),** true**);**

FindPathInRaster**.**SetCell**(**High**(**xRange**),** High**(**yRange**),** true**);**

FindPathInRaster**.**SetCell**((**High**(**xRange**)** **DIV** 2**),** **(**High**(**yRange**)** **DIV** 2**),** true**);**

FindPathInRaster**.**PrintGameField**;**

FindPathInRaster**.**Reset**;**

**END;**

{ Test get Path exists }

**PROCEDURE** TestPathExists**;**

**VAR**

i**,** j**:** INTEGER**;**

top**:** BOOLEAN**;**

**BEGIN**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'PathExists'**);**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**High**(**xRange**),** Low**(**yRange**),** true**);**

WriteLn**(**'Start is wall: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,**Low**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Path exists recursive: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

WriteLn**(**'Path exists iterative: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**Low**(**xRange**),** High**(**yRange**),** true**);**

WriteLn**(**'Goal is wall: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,**Low**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Path exists recursive: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

WriteLn**(**'Path exists iterative: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**5**,** 1**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 2**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 3**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**1**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**2**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**3**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**4**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

WriteLn**(**'Goal is enclosed: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,** 2**,** ', y:'**,** 2**);**

WriteLn**(**'Path exists recursive: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** 2**,** 2**));**

WriteLn**(**'Path exists iterative: '**,** FindPathInRaster**.**PathExists**(**High**(**xRange**),** Low**(**yRange**),** 2**,** 2**));**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**5**,** 1**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 2**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 3**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**1**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**2**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**3**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**4**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

WriteLn**(**'Goal and Start are enclosed: '**);**

WriteLn**(**'Start: x:'**,** 4**,** ', y:'**,** 3**);**

WriteLn**(**'Goal : x:'**,** 2**,** ', y:'**,** 2**);**

WriteLn**(**'Path exists recursive: '**,** FindPathInRaster**.**PathExists**(**4**,** 3**,** 2**,** 2**));**

WriteLn**(**'Path exists iterative: '**,** FindPathInRaster**.**PathExists**(**4**,** 3**,** 2**,** 2**));**

FindPathInRaster**.**PrintGameField**;**

FindPathInRaster**.**Reset**;**

{ Do not Randomize to produce same output }

top **:=** true**;**

**FOR** i **:=** **(**Low**(**xRange**)** **+** 1**)** **TO** **(**High**(**xRange**)** **-** 1**)** **DO** **BEGIN**

**FOR** j **:=** Low**(**xRange**)** **TO** High**(**yRange**)** **DO** **BEGIN**

**IF** **(((**i **mod** 2**)** **<>** 0**)** **AND** **(**top**)** **AND** **(**j **>** Low**(**yRange**)))** **THEN** **BEGIN**

FindPathInRaster**.**SetCell**(**i**,** j**,** true**);**

**END**

**ELSE** **IF** **(((**i **mod** 2**)** **<>** 0**)** **AND** **(NOT** top**)** **AND** **(**j **<** High**(**yRange**)))** **THEN** **BEGIN**

FindPathInRaster**.**SetCell**(**i**,** j**,** true**);**

**END;**

**END;**

**IF** **((**i **mod** 2**)** **<>** 0**)** **THEN** **BEGIN**

**IF** **(**top**)** **THEN** **BEGIN**

top **:=** false**;**

**END**

**ELSE** **BEGIN**

top **:=** true**;**

**END;**

**END;**

**END;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'Find path:'**);**

WriteLn**(**'Start: x:'**,** 1**,** ', y:'**,** 1**);**

WriteLn**(**'Goal : x:'**,** High**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Path exists recursive: '**,** FindPathInRaster**.**PathExists**(**1**,** 1**,** High**(**xRange**),** High**(**yRange**)));**

WriteLn**(**'Path exists iterative: '**,** FindPathInRaster**.**PathExists**(**1**,** 1**,** High**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

FindPathInRaster**.**Reset**;**

**END;**

{ Test get Path exists }

**PROCEDURE** TestShortestPathLength**;**

**VAR**

i**,** j**:** INTEGER**;**

top**:** BOOLEAN**;**

**BEGIN**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

WriteLn**(**'ShortestPathLength'**);**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**High**(**xRange**),** Low**(**yRange**),** true**);**

WriteLn**(**'Start is wall: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,**Low**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Shortest path: '**,** FindPathInRaster**.**ShortestPathLength**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**Low**(**xRange**),** High**(**yRange**),** true**);**

WriteLn**(**'Goal is wall: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,**Low**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Shortest path: '**,** FindPathInRaster**.**ShortestPathLength**(**High**(**xRange**),** Low**(**yRange**),** Low**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

WriteLn**;**

WriteLn**(**'---------------------------------------------------'**);**

FindPathInRaster**.**Reset**;**

FindPathInRaster**.**SetCell**(**5**,** 1**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 2**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 3**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**1**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**2**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**3**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**4**,** 4**,** true**);**

FindPathInRaster**.**SetCell**(**5**,** 4**,** true**);**

WriteLn**(**'Goal is enclosed: '**);**

WriteLn**(**'Start: x:'**,** High**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,** 2**,** ', y:'**,** 2**);**

WriteLn**(**'Shortest path: '**,** FindPathInRaster**.**ShortestPathLength**(**High**(**xRange**),** Low**(**yRange**),** 2**,** 2**));**

FindPathInRaster**.**PrintGameField**;**

FindPathInRaster**.**Reset**;**

WriteLn**(**'Path found '**);**

top **:=** true**;**

**FOR** i **:=** **(**Low**(**xRange**)** **+** 1**)** **TO** **(**High**(**xRange**)** **-** 1**)** **DO** **BEGIN**

**FOR** j **:=** Low**(**xRange**)** **TO** High**(**yRange**)** **DO** **BEGIN**

**IF** **(((**i **mod** 2**)** **<>** 0**)** **AND** **(**top**)** **AND** **(**j **>** Low**(**yRange**)))** **THEN** **BEGIN**

FindPathInRaster**.**SetCell**(**i**,** j**,** true**);**

**END**

**ELSE** **IF** **(((**i **mod** 2**)** **<>** 0**)** **AND** **(NOT** top**)** **AND** **(**j **<** High**(**yRange**)))** **THEN** **BEGIN**

FindPathInRaster**.**SetCell**(**i**,** j**,** true**);**

**END;**

**END;**

**IF** **((**i **mod** 2**)** **<>** 0**)** **THEN** **BEGIN**

**IF** **(**top**)** **THEN** **BEGIN**

top **:=** false**;**

**END**

**ELSE** **BEGIN**

top **:=** true**;**

**END;**

**END;**

**END;**

WriteLn**;**

WriteLn**(**'Start: x:'**,** Low**(**xRange**),** ', y:'**,** Low**(**yRange**));**

WriteLn**(**'Goal : x:'**,** High**(**xRange**),** ', y:'**,** High**(**yRange**));**

WriteLn**(**'Shortest path: '**,** FindPathInRaster**.**ShortestPathLength**(**Low**(**xRange**),** Low**(**yRange**),** High**(**xRange**),** High**(**yRange**)));**

FindPathInRaster**.**PrintGameField**;**

**END;**

**BEGIN**

TestReset**;**

TestSetCell**;**

TestPathExists**;**

TestShortestPathLength**;**

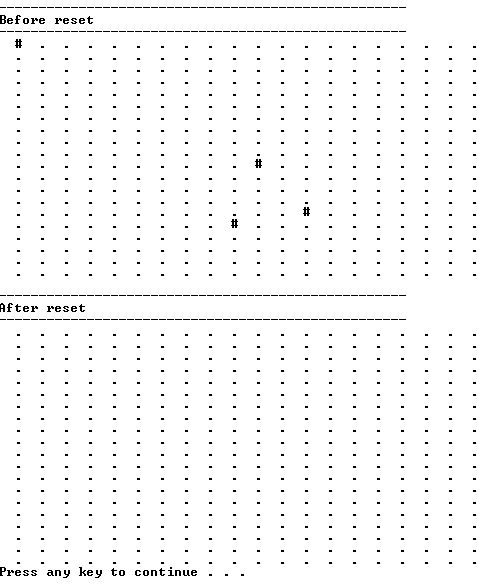
**END.**

## Tests

Folgend sind Tests der FindPathInRasterUnit angeführt.

### Reset / PrintGameField

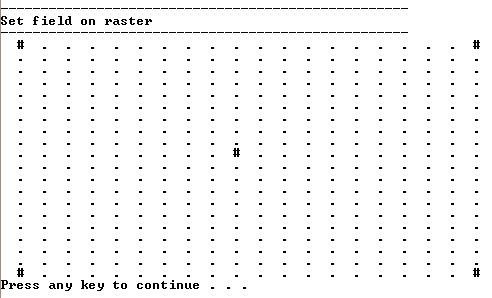
Dieser Test testet die Prozedur Reset, die das Gedächtnis (Spielfeld) zurücksetzt, in dem sie ein leeres Spielfeld. Gleichzeitig wird hierbei auch die Prozedur PrintGameField getestet, die das Spielfeld auf der Konsole ausgibt.



Nachdem Felder auf dem Speilfeld platziert wurden und die Prozedur Reset aufgerufen wurde, ist zu sehen, das das Spielfeld zurückgesetzt wurde.

### SetCell

Dieser Test testet die Prozedur SetCell, mit der Felder am Spielfeld gesetzt werden können.



Hierbei ist zu sehen, dass die Wände am Spielfeld entsprechend der definierten Koordinaten gesetzt wurden. (Siehe Test Source)

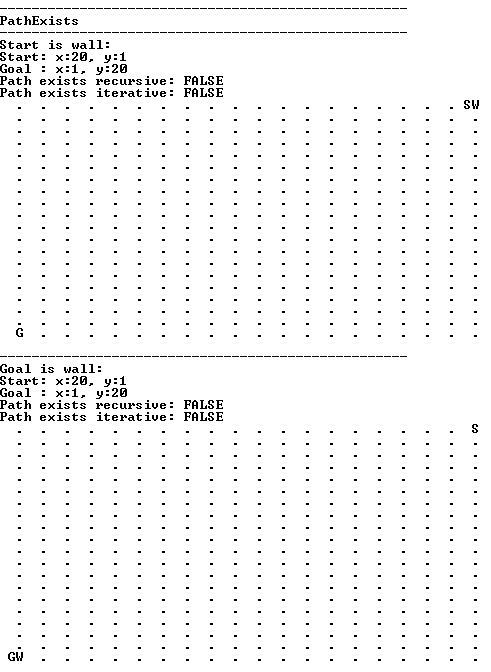
### PathExists

Dieser Test testet die Prozedur PathExists.

Folgende Symbole sind wie folgt zu interpretieren:

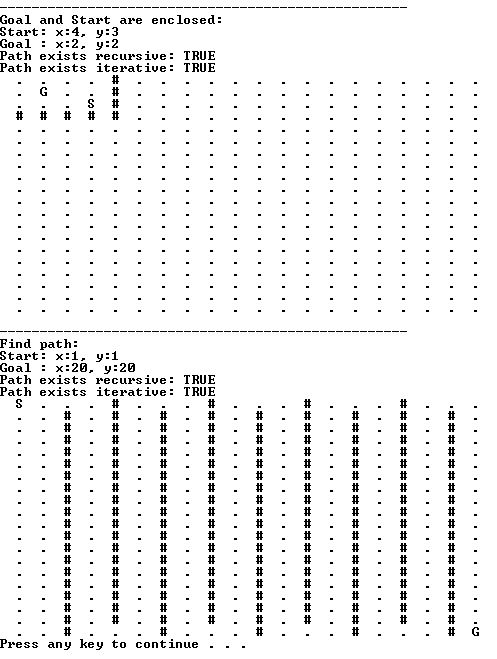
1. S = Start
2. G = Goal
3. SW = Start ist eine wand
4. GW = Goal ist eine Wand

Kein Pfad existiert:



Sollte kein Pfad existieren, so wird als Resultat FALSE zurückgeliefert. Dies erfolgt wen entweder der Start oder das Ziel eine Wand sind oder wenn das Ziel nicht erreicht werden kann.

Pfad ist vorhanden:

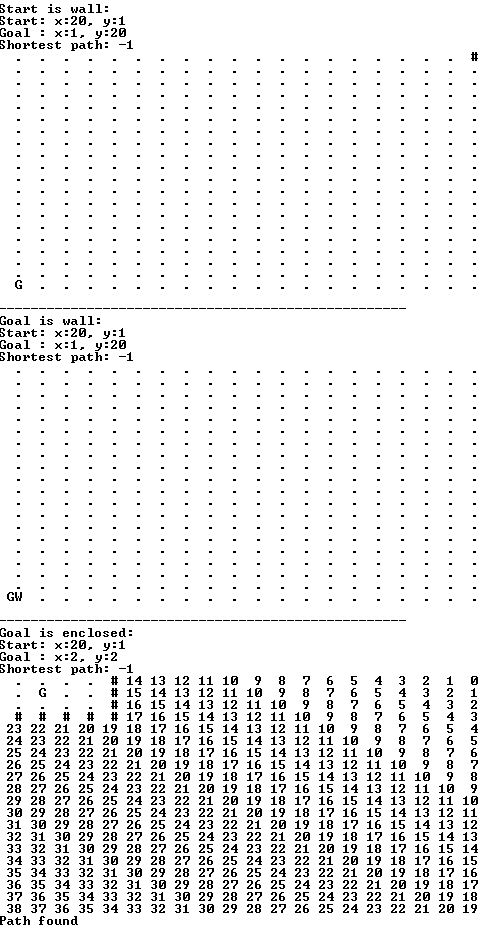


Sollte ein Pfad vorhanden sein so wird als Resultat TRUE zurückgeliefert.

### LengthOfShortestPath

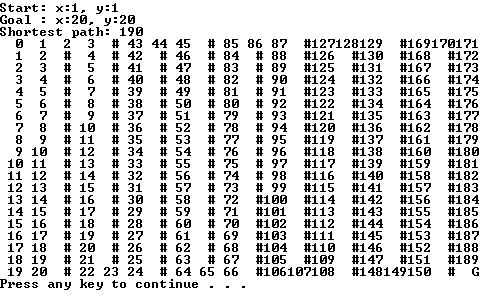
Dieser Test testet die Prozedur LengthOfShortestPath.

Kein Pfad vorhanden:



Hierbei ist zu sehen, dass wenn der Start oder das Ziel eine Wand sind kein Pfad ermitteln kann. Sollte das Ziel von Wänden eingeschlossen sein, so ist zu sehen, wie der Algorithmus sich dem Ziel annähert.

Pfad vorhanden:



Hierbei ist zu sehen wie sich der Algorithmus sich dem Ziel erfolgreich annähert und auch das Ziel erreichen kann.