Theorie III für Softwaretechnik:

5. Backtracking

Gliederung:

5.0 Übersicht

Um Probleme zu lösen, entwickelt man Algorithmen. Um gute Lösungen zu finden, müssen die Algorithmen eine gewisse Qualität aufweisen. Hierzu muss man auf einige Dinge achten:
- Vergleich des Problems mit verwandten Problemen,
- Untersuchung von einfacheren Spezialfällen,
- Einsetzen gängiger Entwicklungsmethoden,
- Analyse der Algorithmen bzgl. der Laufzeit,
- Analyse der Algorithmen bzgl. des Speicherplatzes,
- Untersuchung sonstiger Eigenschaften.

Wer Algorithmen entwickeln und einsetzen will, muss auf diese Punkte achten. Im Mittelpunkt steht meist die Entwicklungsmethode. Bisher haben wir "Greedy", "Dynamisches Programmieren" und "Divide and Conquer" behandelt. Nun stellen wir die Methode des systematischen Durchtestens aller möglichen Lösungskandidaten vor. Diese Methode ist unter dem Namen **Backtracking** geläufig.

Am Ende dieses Abschnitts soll diese Methode des Backtracking gut verstanden und an mehreren Beispielproblemen vermittelt worden sein.

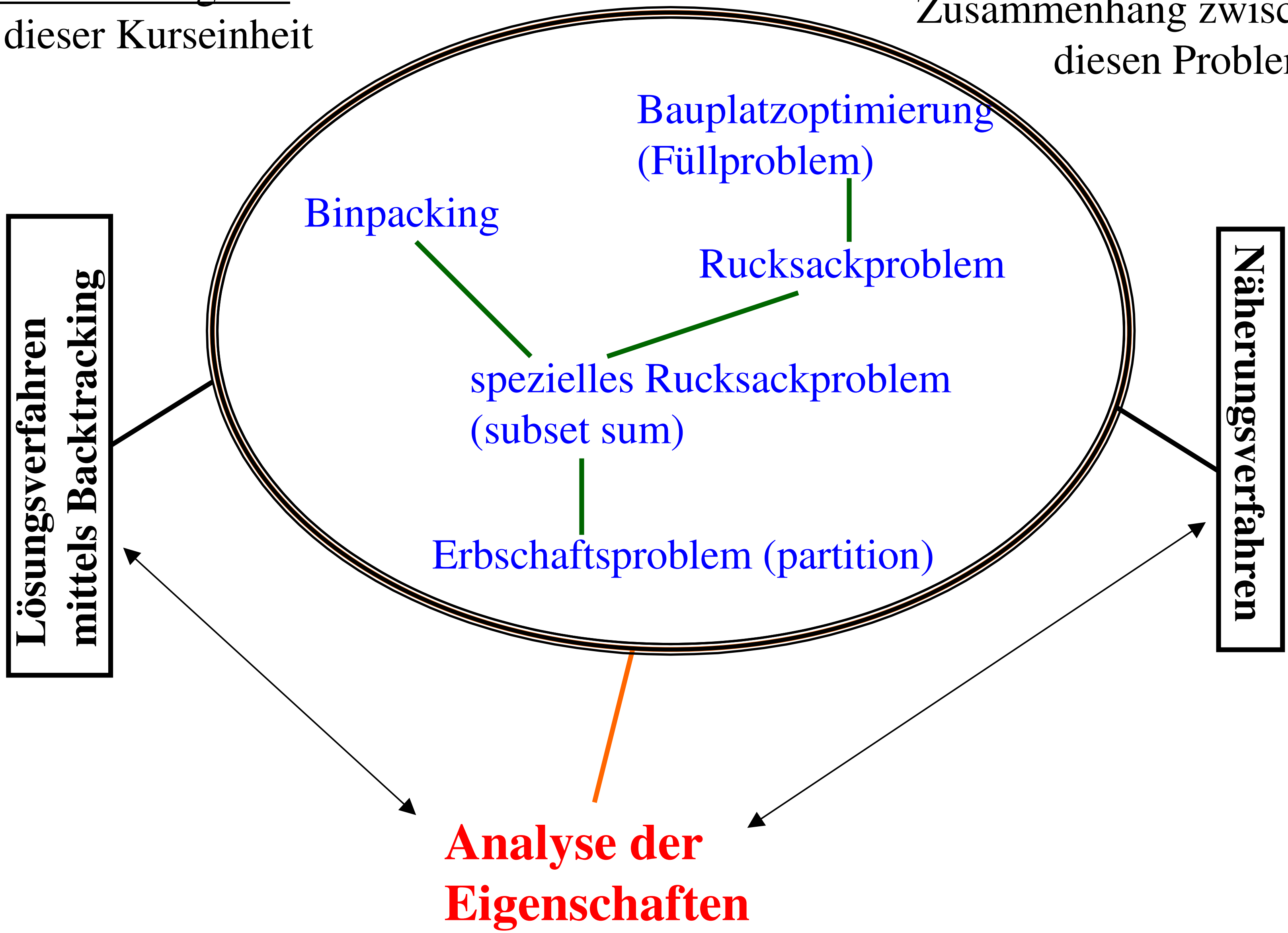
Zum Vorgehen: Zunächst betrachten wir ein "Alltagsproblem", und zwar das optimale Platzieren von Bauplätzen auf einem quadratischen Baugrundstück. Wichtig ist es, ein formales Modell (in Form von präzisen Definitionen) hieraus abzuleiten. Anschließend vereinfachen wir dieses auf Flächen (also auf zwei Dimensionen) bezogene Problem auf den eindimensionalen Fall und gelangen zum Rucksackproblem und zum Erbschaftsproblem. Für dessen Lösung ist das Backtracking unmittelbar einsichtig. Danach verkomplizieren wir das Erbschaftsproblem zum Auffüllproblem für mehrere Behälter (Binpacking), das nun relativ leicht mittels Backtracking gelöst werden kann. Dann erst bearbeiten wir das Ausgangsproblem der optimalen Bauplätze. Die Analyse wird ergeben, dass diese Verfahren in der Praxis nur beschränkt einsetzbar sind. Daher diskutieren wir Näherungsverfahren und deren Vor- und Nachteile.

Mit dieser Kurseinheit werden mehrere Ziele verfolgt. Die Hörer(innen) sollen an konkreten Beispielen lernen

- wie man ein Beispiel vorstellt (5.1),
- wie man hieraus ein Problem modelliert/definiert
- und wie man hierbei vorgehen sollte (5.2),
- wie man Unter-/Teilprobleme abspaltet
- und wie man diese präzise definiert (5.3),
- wie man das Lösungsverfahren Backtracking hierauf anwendet
- und wie man es als Programm schreibt (5.4)
- wie man es auf ein weiteres Problem (BPP) überträgt,
- wie man dessen Eigenschaften untersucht und vor allem
- wie man Laufzeit und Speicherbedarf analysiert (5.5),
- wie man nun das Ausgangsproblem löst (5.6).

Skizze der Begriffe
in dieser Kurseinheit

Zusammenhang zwischen
diesen Problemen



5.1 Einen Bauplatz optimal planen

Ein Gelände wird zur Bebauung freigegeben und soll nun erschlossen werden. Hierfür muss die Gesamtfläche in einzelne Bauplätze (Parzellen, Häuser) und in Straßen (Wege) aufgeteilt werden.

Wir vereinfachen das Problem, indem wir annehmen:
1. Quadratisches Gelände, Seitenlänge: n Maßeinheiten
(z.B.: ist die Maßeinheit 4 Meter, dann ist das Gesamtgelände bei n=20 genau $80 \times 80 = 6400$ Quadrameter groß).
2. Wege bestehen aus 1 Quadrat-Maßeinheiten.
3. Jeder Bauplatz, später „Haus“ genannt, ist ein Vielfaches von Quadrat-Maßeinheiten. Bauplätze müssen sich an vorgegebene Muster halten, in der Praxis meist rechteckig und nicht zu schmal.

Nur durch den Verkauf der Bauplätze kann der Eigentümer Geld verdienen. Er wird also bemüht sein, die Bauplätze so anzuordnen, dass möglichst wenig Verschnitt übrig bleibt, dass also der Anteil, der für Wege verwendet werden muss, möglichst gering ist. Zugleich sollen die Bauplätze von einem vorgegebenen Startfeld S (dies ist in der Regel die verkehrsmäßige Anbindung an das bereits bestehende Straßennetz) im Mittel schnell erreicht werden können. Es sollen also keine schlangenlinienartigen Wege, sondern gut verzweigte Straßen zu den Bauplätzen führen; insbesondere soll es keine zu langen Wege innerhalb des Gesamtgeländes geben.

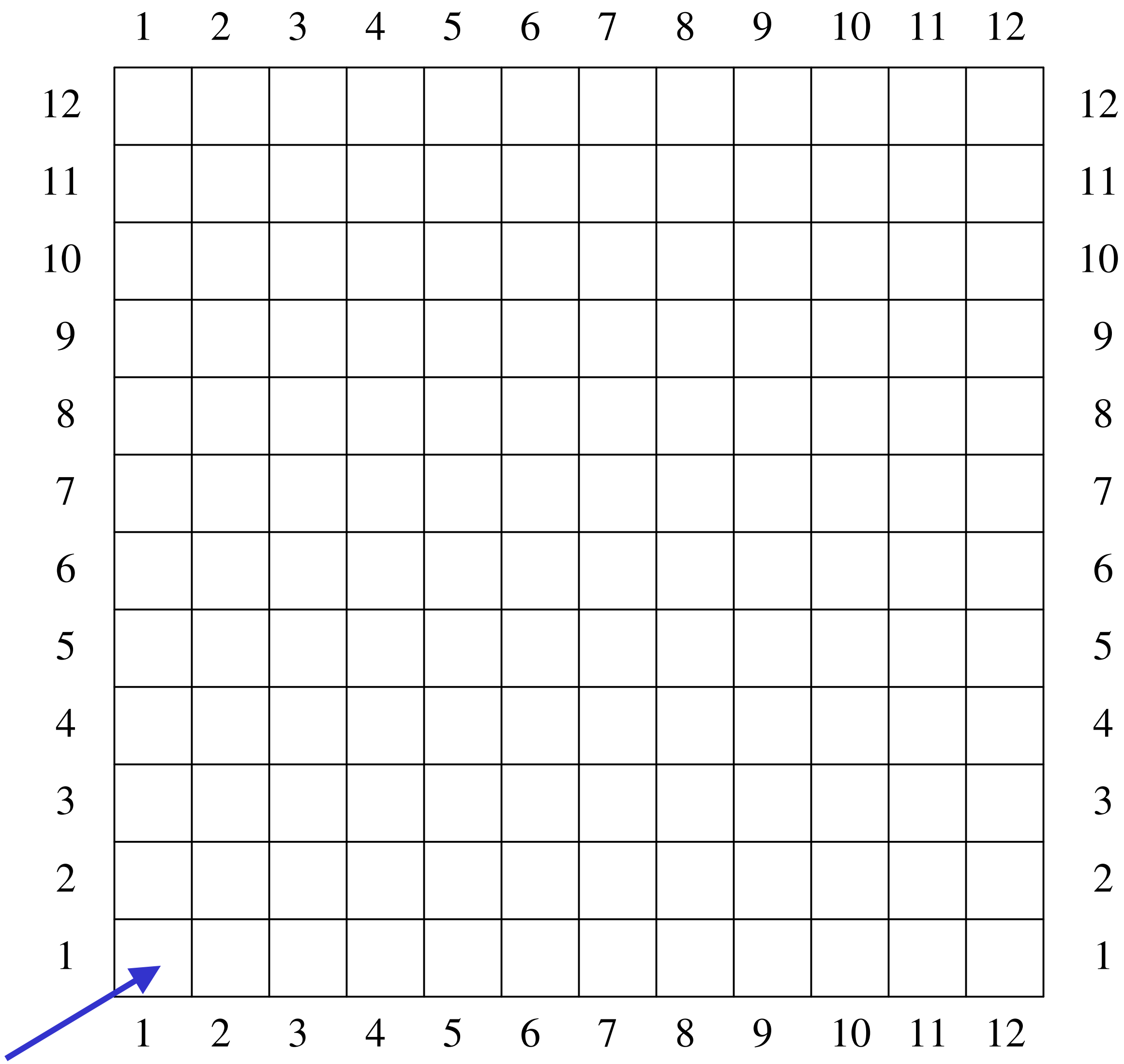
Wir erläutern das Problem und die hierbei auftretenden Parameter an einem Beispiel (n=12, Bauplatzgröße m=6).

Aufgabe:

Platzieren Sie auf einer $n \times n$-Gesamtfläche zu einem vorgegebenen Startfeld S möglichst viele Bauplätze (= Häuser = rechteckige Flächen der Größe m) und Zugangswege (bestehend aus rechteckig aneinander gesetzten Einheiten der Größe 1), so dass jedes Haus von S aus über die Zugangswege erreichbar ist und der mittlere Abstand μ von S zu allen Häusern möglichst gering ist.

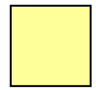
Schauen wir uns ein Beispiel an (n=12, m=6):

Die Gesamtfläche G = 12×12 = 144 (Felder)

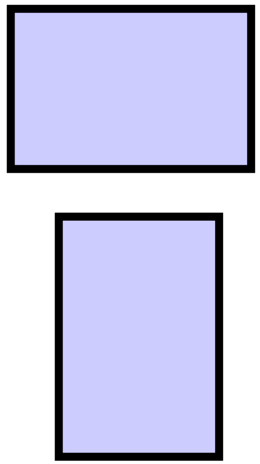


Start **S** = Feld (1,1)

Weg-Einheit:

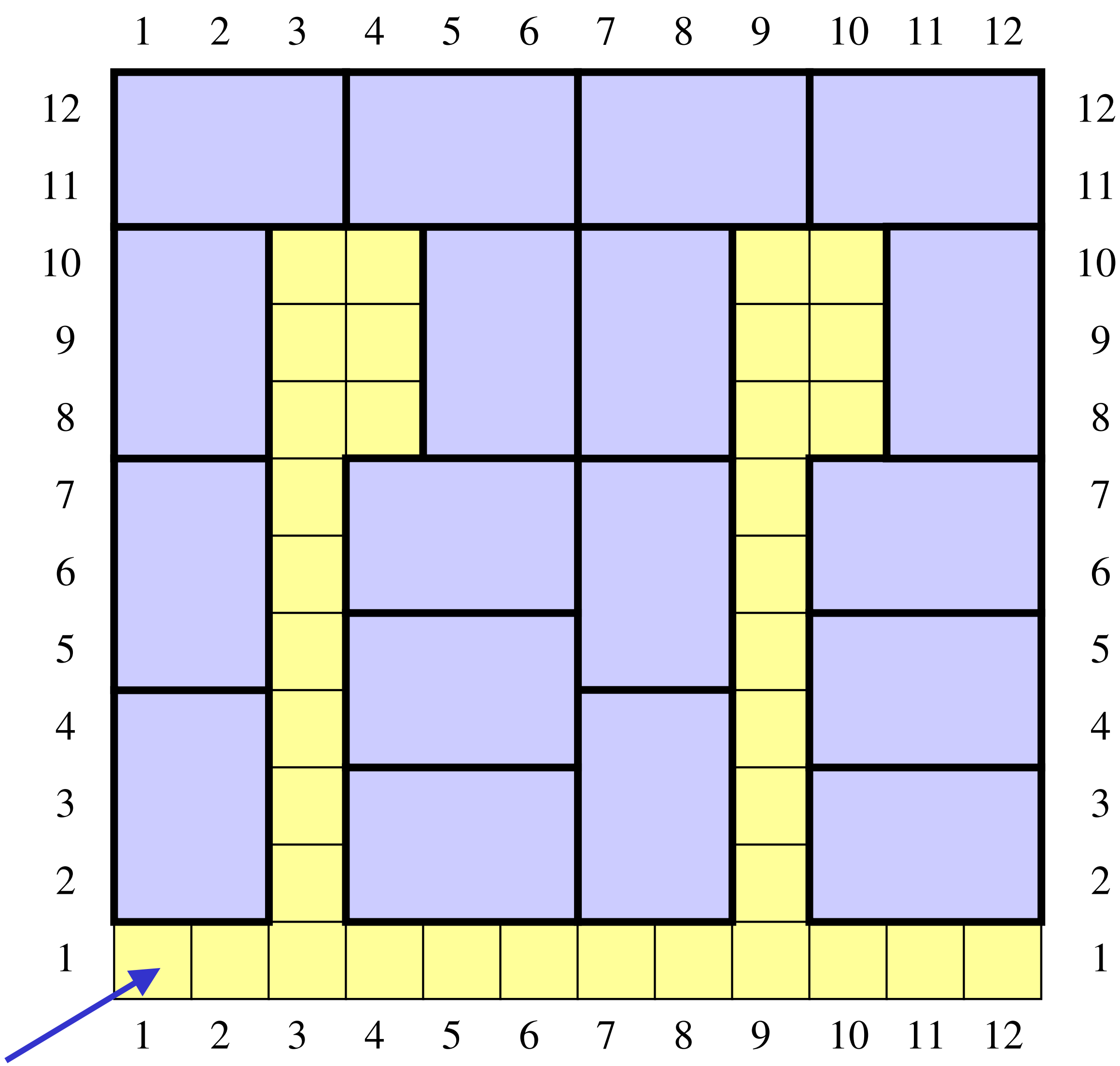


Zulässige Muster für die Häuser:



Hinweis: Im ersten Schritt muss das Feld **S** betreten werden.

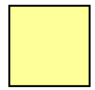
Lösungsvorschlag 1: Wir setzen Häuser irgendwie in die Gesamtfläche ein:



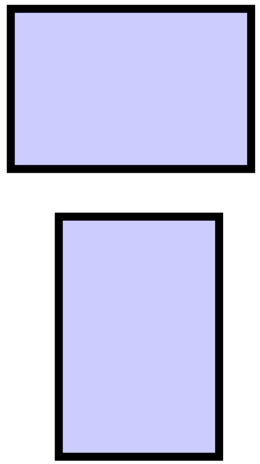
Start **S** = Feld (1,1)

Jetzt noch die
Wege hervorheben.

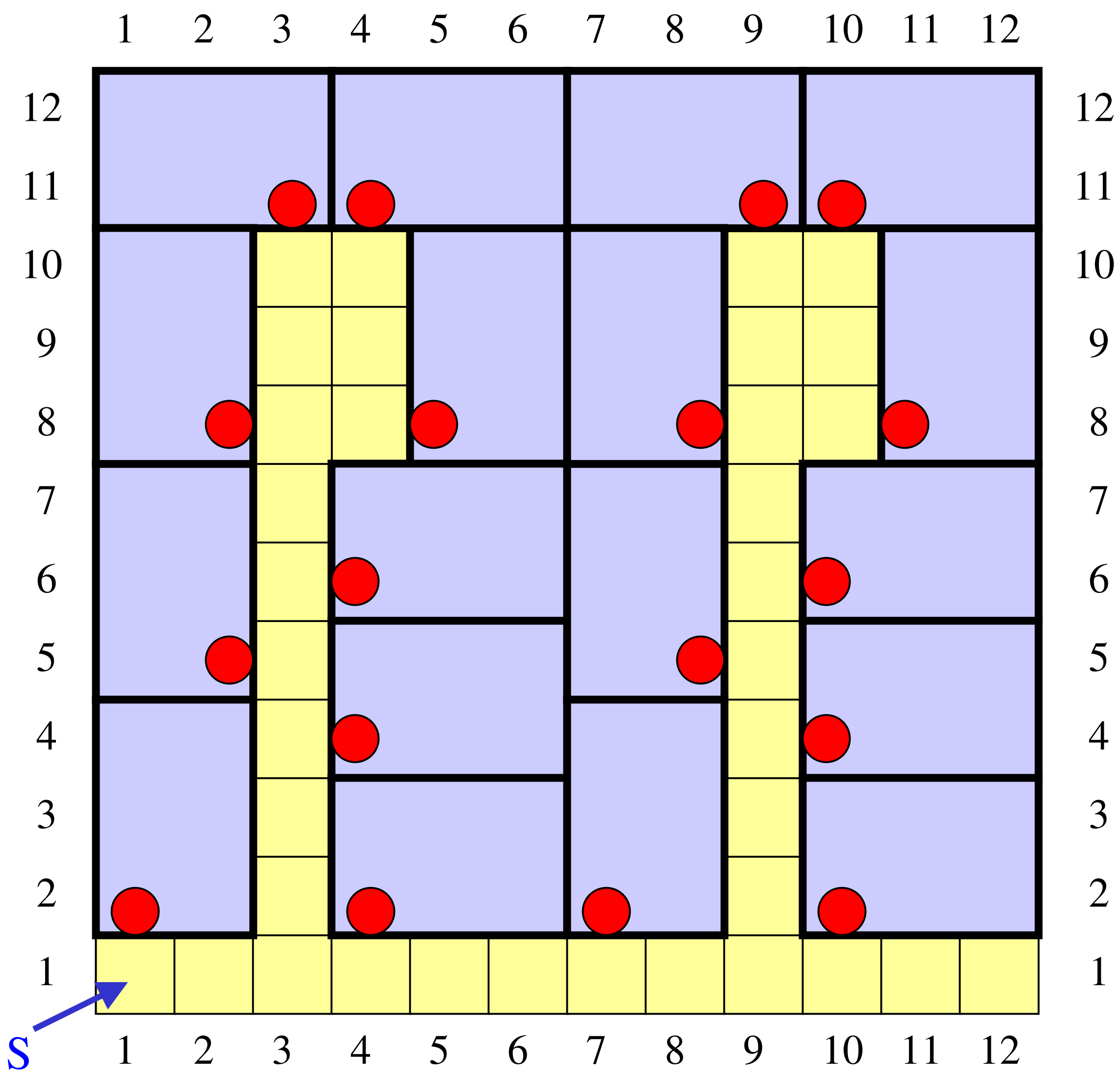
Weg-Einheit:



Zulässige Muster
für die Häuser:



Zum Lösungsvorschlag 1: Mittleren Abstand μ berechnen:



Werte in diesem Beispiel:

n=12
Gesamtfläche G=144
m=6 (Muster: 2×3, 3×2)
S = (1,1) Startfeld
W=36 (Felder) für Wege (gelb)
H=18 (Zahl der Häuser)
Kontrolle: Es gilt
G = H×m + W, also
144 = 18×6 + 36

Lassen sich mehr als 18 Häuser unterbringen?

Ja, siehe später.

Legt man den Eingang immer optimal nahe zu S (roter Punkt!), so folgt als mittlerer Abstand:
μ = (1+4+6+7+7+8+10+10+11+12+12+13+13+14+16+17+18+19) / 18 = 198/18 = 11.

Bezeichnungen und Vereinbarungen: Wir beschreiben alles in der Ebene der natürlichen Zahlen, wobei die Felder durch (i,j) mit $1 \leq i \leq n$ und $1 \leq j \leq n$ bezeichnet werden. Als Startfeld wählen wir zunächst S=(1,1). Wir nehmen stets an, dass das Feld S anfangs im ersten Schritt von außen betreten wird. H sei die Anzahl der Häuser (sie belegen jeweils eine Fläche der Größe m, insgesamt also eine Fläche $H \cdot m$) und W sei die Zahl der Felder, die für Wege benutzt werden. Es sei μ die mittlere Zugangszeit oder mittlere Weglänge, gemessen als mittlerer Abstand von S zu irgendeinem Haus, d.h.: μ ist die Summe der Wegeeinheiten von allen Häusern auf einem kürzesten Weg zu S dividiert durch die Anzahl H der Häuser:

$$\mu = (\text{Abstand}_1 + \text{Abstand}_2 + \ldots + \text{Abstand}_H) / H ,$$

wobei Abstand_i der (kürzeste) Abstand des Hauses i vom Startfeld S ist.

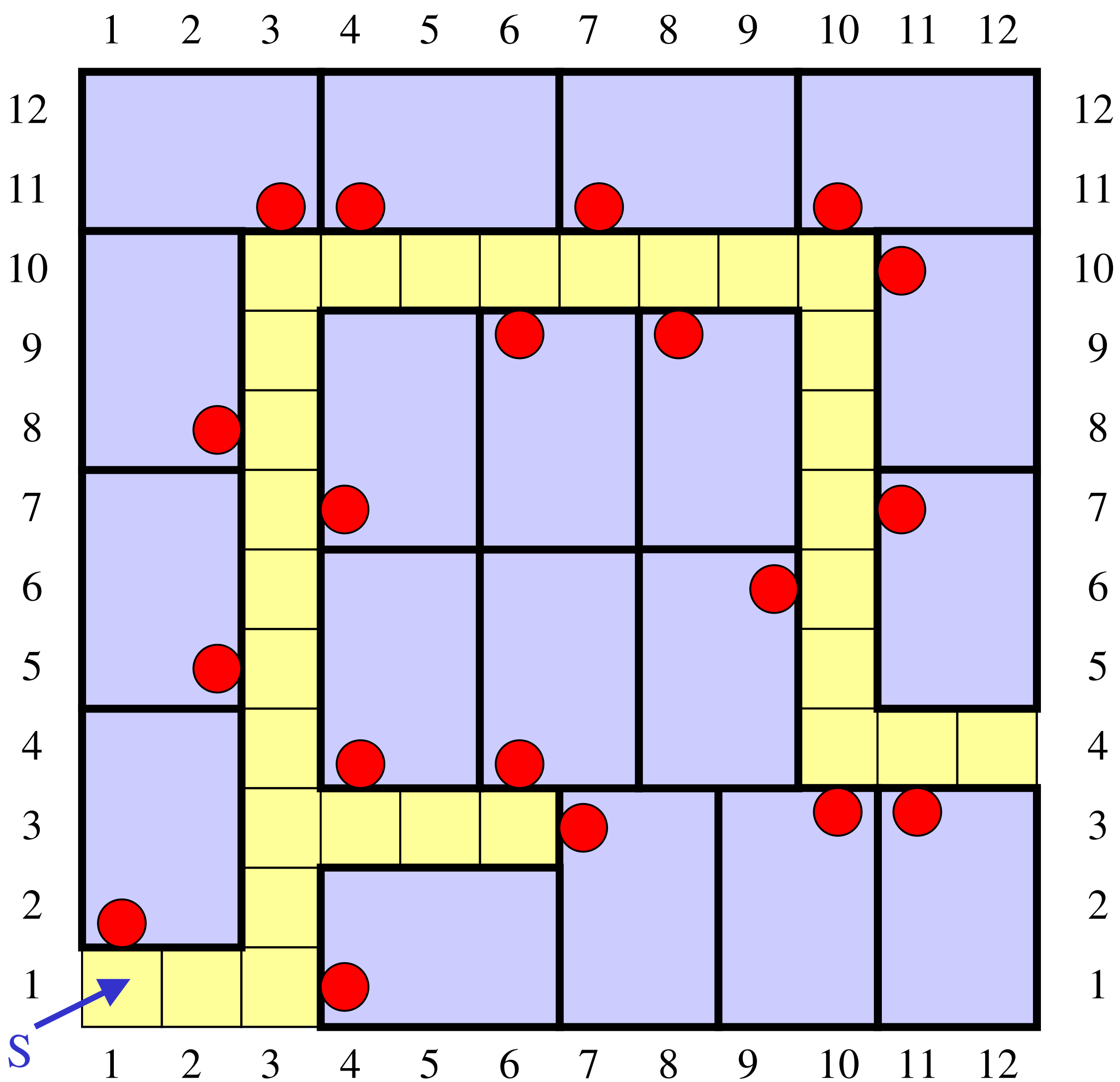
Aufgabe: Finden Sie weitere Lösungen, z.B. mit mehr Häusern oder mit kleinerem mittlerem Abstand μ.

Hinweis: Wenn n=12 ist, dann hat die Gesamtfläche 144 Felder. Wegen 144/6=24 und weil Wege von S zu allen Häusern führen müssen, kann man vermutlich nur mit höchstens 22 Häusern rechnen.

Auf den folgenden Folien finden Sie weitere Beispiele für n=12, m=6 (wobei als "Häuser" nur die Muster mit 2×3 und 3×2 Feldern zugelassen sind) und für verschiedene Startfelder.

Lösen Sie die Aufgabe zunächst alleine.

Lösungsvorschlag 2:



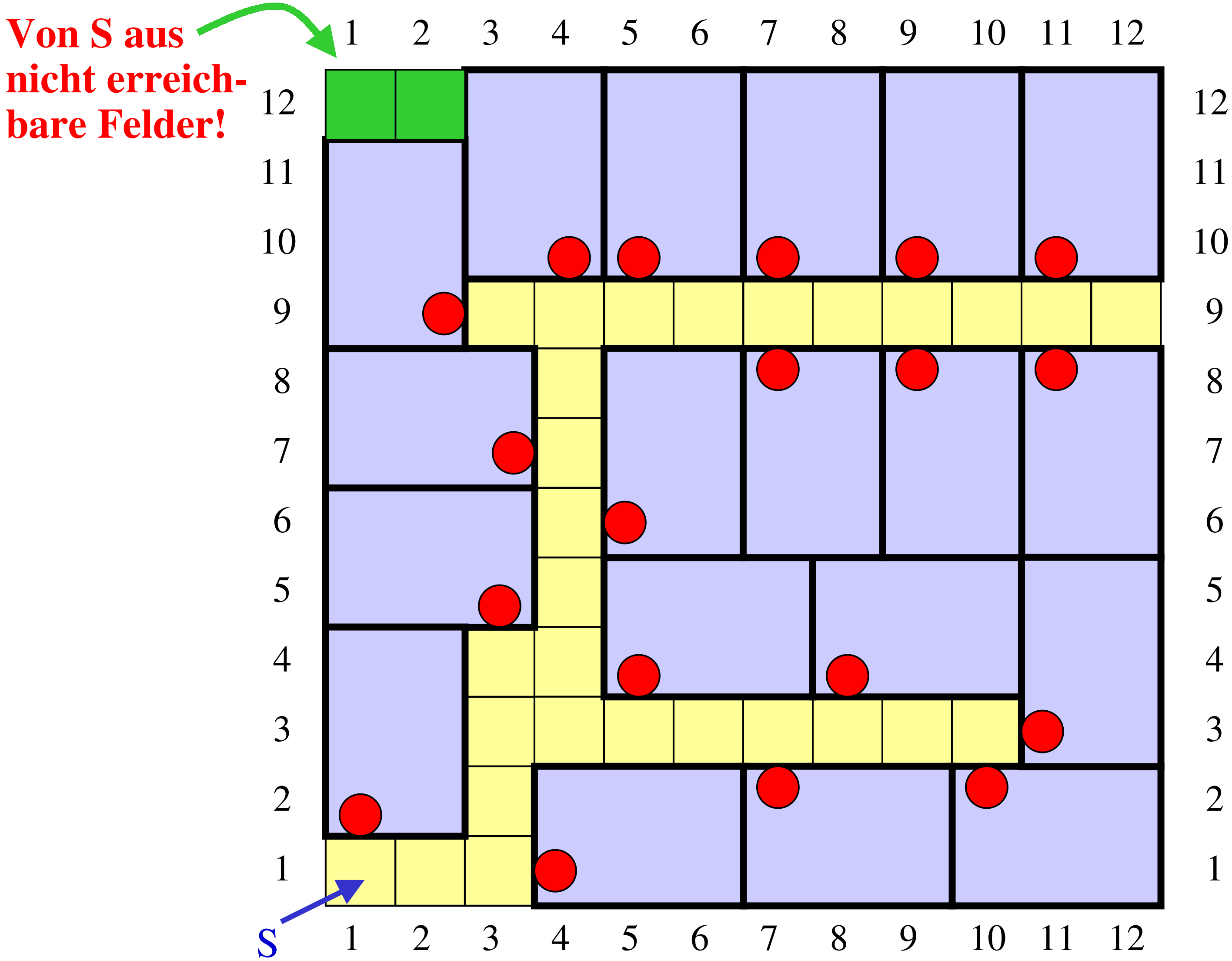
n=12
Gesamtfläche
G=144
m=6 (2×3 und 3×2)
S=(1,1)
W=30 (Felder) für
Wege (gelb)
H=19 (Zahl der
Häuser)
Kontrolle:
G = H×m + W, also
144 = 19×6 + 30

Kann man μ verringern, ohne die Zahl der Häuser zu verkleinern?

Ja, siehe nächster Lösungsvorschlag

Legt man den Eingang immer optimal nahe zu S (roter Punkt!), so folgt:
$\mu = (1+3+6+7+8+8+9+10+12+13+15+16+17+19+19+22+23+25+26) / 19 = 259/19 \approx 13{,}63$.

Lösungsvorschlag 3:



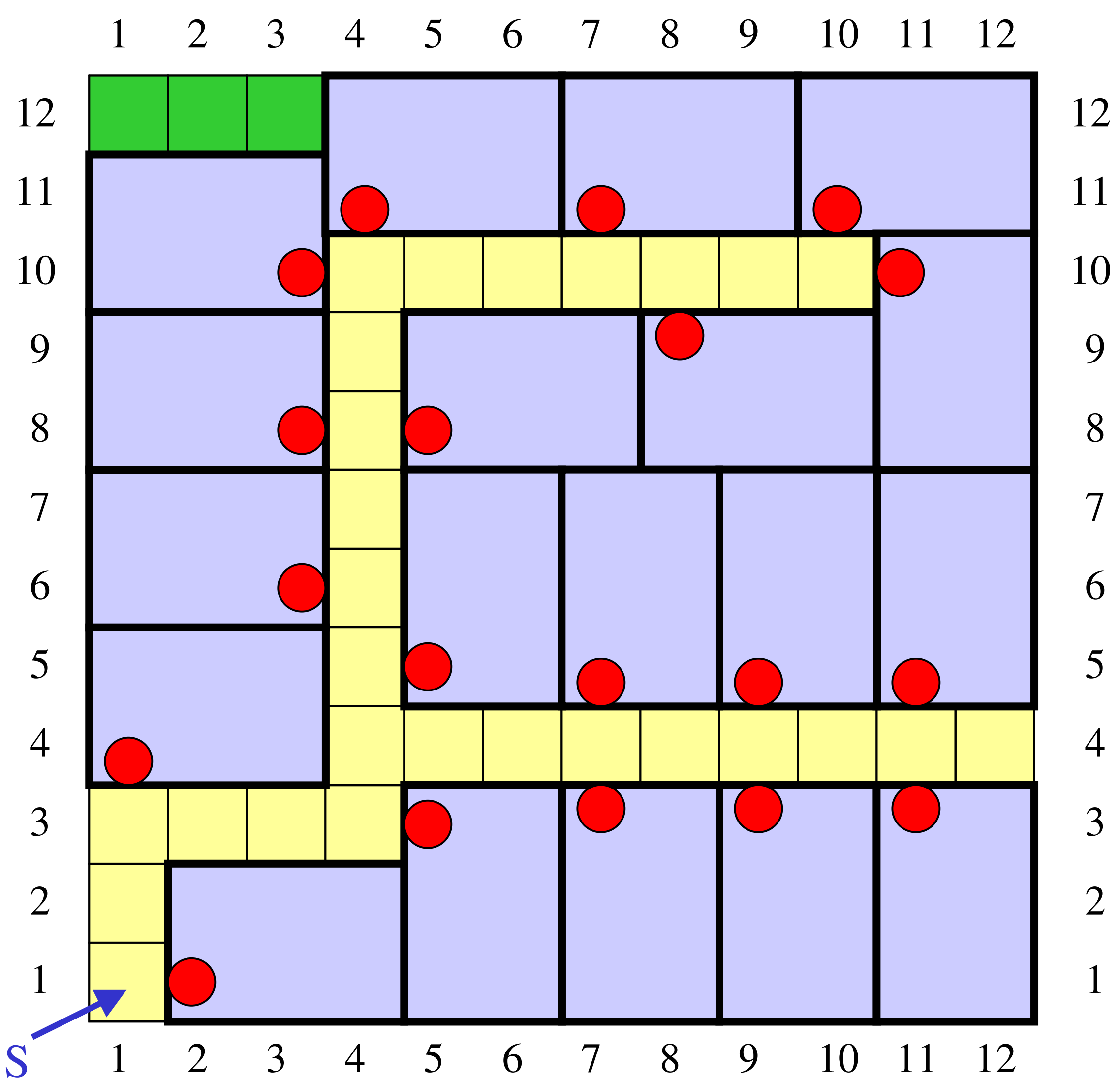
n=12
Gesamt: G=144
m=6 (2×3 und 3×2)
S=(1,1)
W=28 (Wegefelder)
U=2, Zahl der von S nicht erreichbaren Felder (grün)
H=19 (Häuser)
Kontrolle:
G = H×m + W + U,
144 = 19×6+28+2

Kann man μ weiterhin verringern, ohne die Zahl der Häuser zu verkleinern?

Ja, siehe nächster Lösungsvorschlag.

Legt man den Eingang immer optimal nahe zu S (roter Punkt=Eingang des Hauses), so folgt:
$\mu = (1+3+6+7+9+9+10+10+12+12+12+13+13+15+15+17+17+19+19) / 19 = 219/19 \approx 11{,}53.$

Lösungsvorschlag 4:



n=12
Gesamt: G=144
m=6 (2×3 und 3×2)
S=(1,1)
W=27 (Wegefelder)
U=3, Zahl der von S nicht erreichbaren Felder
H=19 (Häuser)
Kontrolle:
$G = H \times m + W + U$,
$144 = 19 \times 6 + 27 + 3$

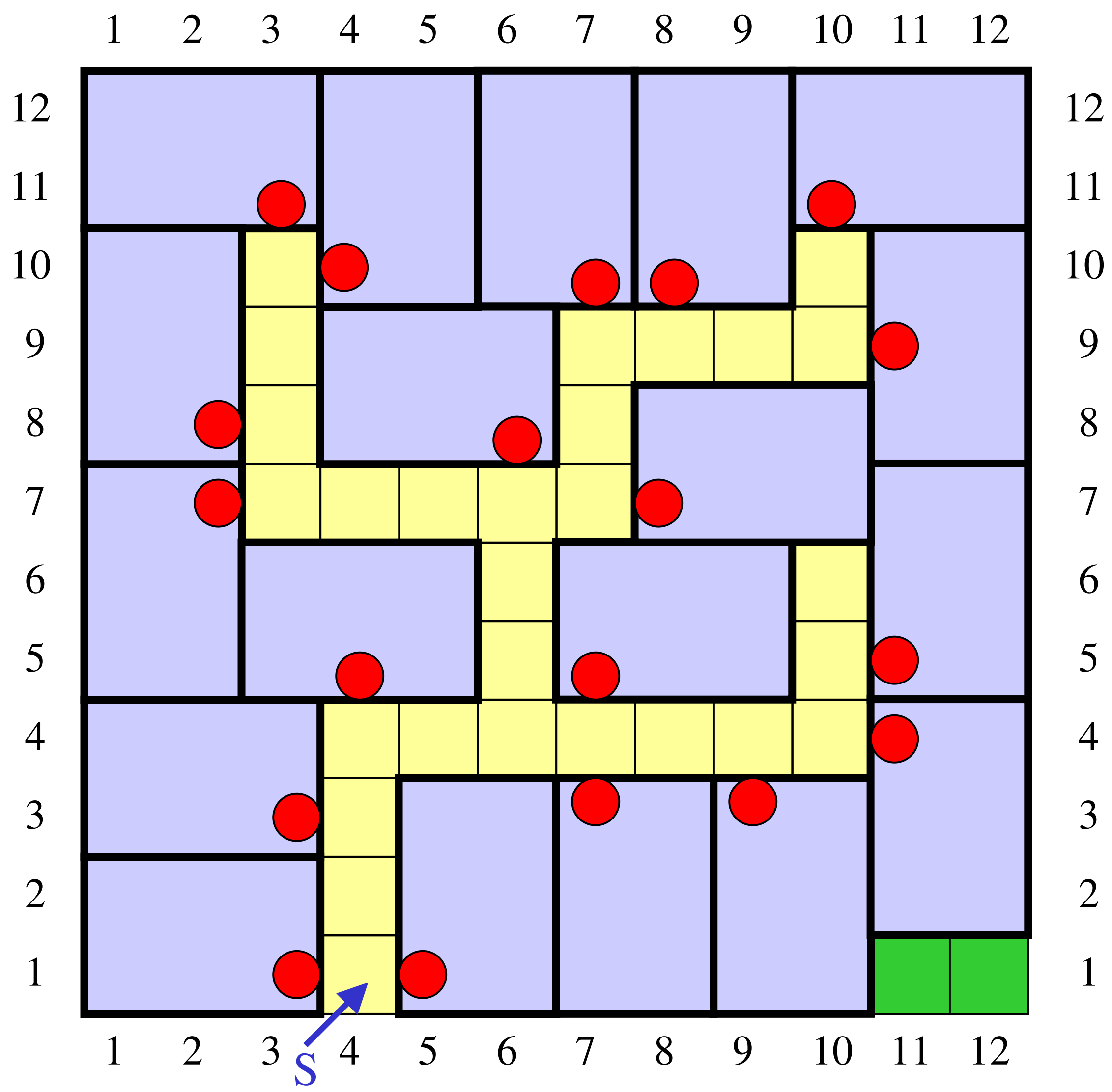
Kann man μ weiterhin verringern, ohne die Zahl der Häuser zu verkleinern?

Ja, selbst suchen.

Wir ändern nun die Lage von S.

Legt man den Eingang immer optimal nahe zu S (rote Punkte), so folgt:
$\mu = (1+3+6+8+9+10+10+11+11+12+12+13+13+14+14+16+17+19+19) / 19 = 218/19 \approx 11{,}47$.

Lösungsvorschlag 5 , neues Startfeld (4,1):



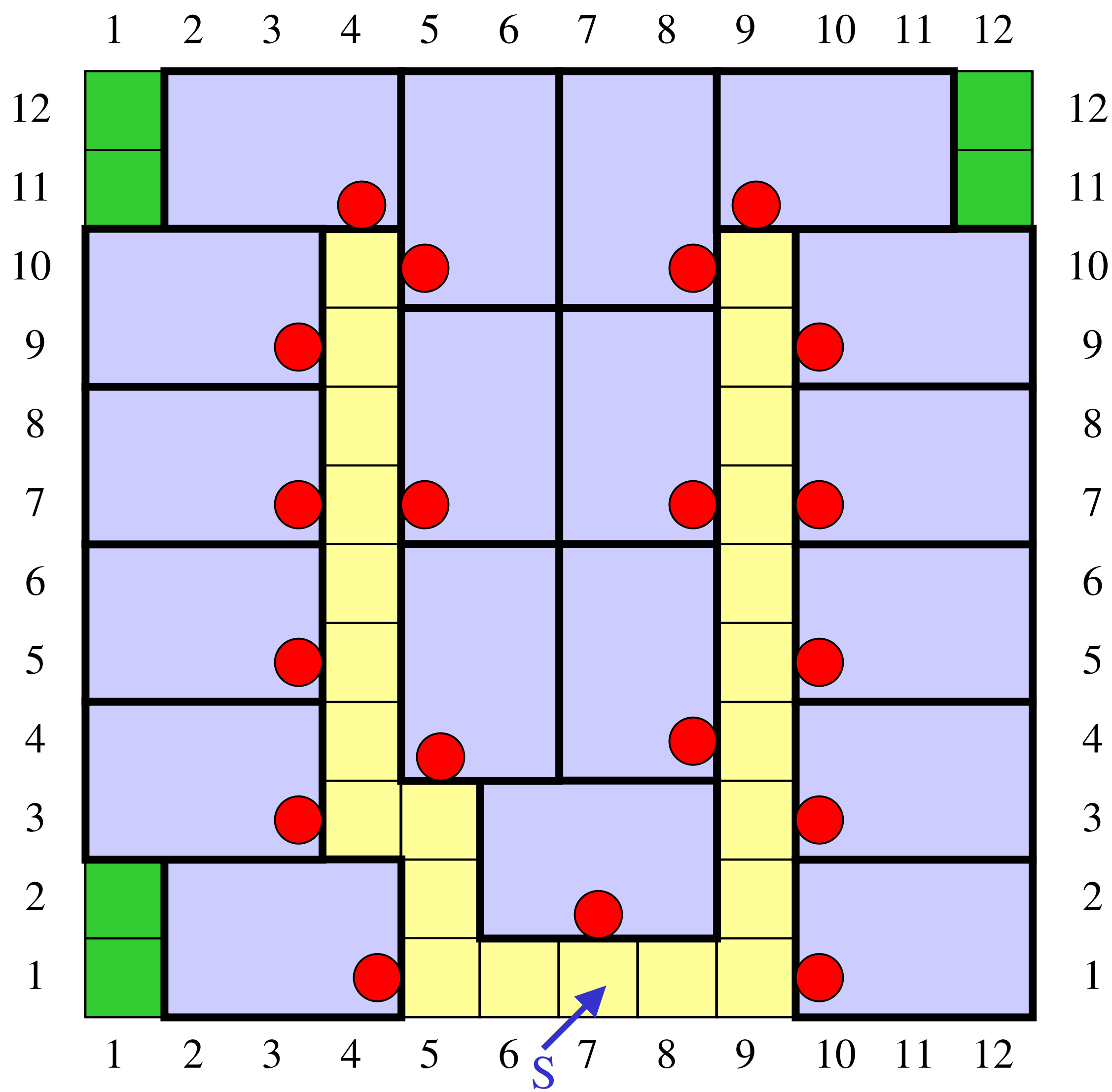
n=12
Gesamt: G=144
m=6 (2×3 und 3×2)
S=(4,1)
W=28 (Wegefelder)
U=2 (unerreich-
bare Felder)
H=19 (Häuser)
Kontrolle:
G = H×m + W + U,
144 = 19×6+28+2

Kann man μ weiterhin verringern?
Ja.
Selbst suchen.

Wir legen nun S in die Mitte des Randes: S= (7,1).

Legt man den Eingang immer optimal nahe zu S (rote Punkte), so folgt:
μ = (1+1+3+4+7+7+9+9+10+10+11+12+12+13+13+15+15+15+16) / 19 = 183/19 ≈ 9,63.

Lösungsvorschlag 6, neues Startfeld (7,1):



n=12
Gesamt: G=144
m=6 (2×3 und 3×2)
S=(7,1)
W=24 (Wegefelder)
U=6 (unerreich-
bare Felder)
H=19 (Häuser)
Kontrolle:
G = H×m + W + U,
144 = 19×6+24+6

Durch kleine Veränderungen kann man oft μ etwas verbessern. So auch hier: Verschiebe den Mittelteil nach unten und schaffe dessen untere Fläche nach oben.

Legt man den Eingang immer optimal nahe zu S (rote Punkte), so folgt:

μ = (1+3+3+5+5+6+6+7+8+9+9+10+10+11+12+12+12+13+13) / 19 = 155/19 ≈ 8,16.

Frage:

Wenn in diesem Beispiel das Startfeld S irgendwo läge, wie viele Felder für Wege braucht man mindestens?

Anders gefragt: Wie viele Häuser kann man maximal auf der Gesamtfläche unterbringen, wenn das Startfeld keine Rolle spielt, sondern nur gefordert wird, dass alle Häuser untereinander durch Wege verbunden sind?

Antwort:

Wir kennen die Antwort nicht. Nach längerem Ausprobieren vermutet man, dass es keine Lösung mit 20 Häusern gibt. (Selbst probieren. Eine Lösung mit 21 oder mehr Häusern kann es nicht geben, siehe später.)

Frage:

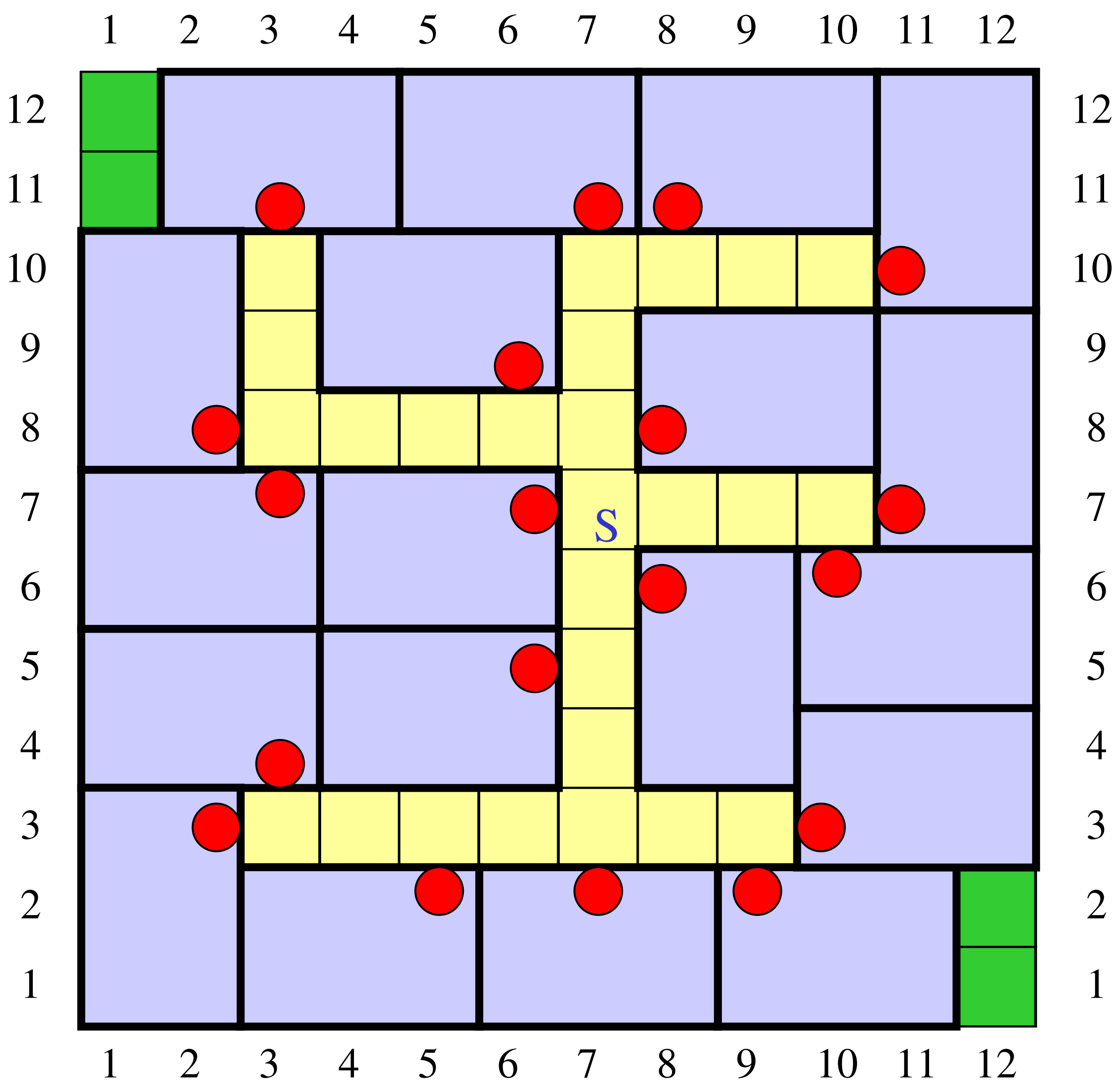
Wo ist die optimale Lage eines Startfeldes S, damit bei 19 Häusern der mittlere Abstand μ minimal wird?

Antwort:

Wir kennen auch hier die Antwort nicht. Wir vermuten stark, dass S in der Mitte liegen muss, also S = (7,7). Hierfür haben wir eine Lösung mit μ=5,21 gefunden, die auf der nächsten Folie vorgestellt wird. Diese Lösung hat keine Öffnung nach außen, sie könnte in einer praktischen Anwendung z.B. die optimale Lage einer U-Bahnstation angeben.

Vielleicht finden Sie eine bessere Lösung?

Lösungsvorschlag 7 (möglichst kleines μ bei 19 Häusern):



n=12
Gesamt: G=144
m=6 (2×3 und 3×2)
S = (7,7)
W=26 (Wegefelder)
U=4 (unerreich-
bare Felder)
H=19 (Häuser)
Kontrolle:
G = H×m + W + U,
144 = 19×6+26+4

Legt man den Eingang immer optimal nahe zu S (rote Punkte), so folgt:
μ = (1+2+2+3+3+4+4+4+5+5+6+6+7+7+7+7+8+9+9) / 19 = 99/19 ≈ 5,21.

Leider gewinnt man auch nach längerem Probieren und Studieren kaum eine Einsicht in die Struktur dieser Problemklasse. Es bleiben viele Fragen offen, z.B.:

1.) Ein effizientes systematisches Verfahren konnten wir aus den Beispielen nicht ableiten, vielmehr haben wir einfach nur einige Lösungen ausprobiert. Allgemein kennt man außer dem vollständigen Durchprobieren aller Lösungen bisher auch kein gutes Verfahren, das mit Sicherheit eine optimale Lösung findet. Man vermutet, es gibt es auch keine schnellen Verfahren, um eine oder alle optimalen Lösungen für das hier betrachtete Füllproblem zu finden. (Stichwort: NP-harte Probleme.)

2.) Wir haben keine <u>Beweise</u> angegeben, um sinnlose Versuche zu vermeiden oder um untere Schranken für μ oder obere Schranken für H anzugeben. Auch hier kennt man keine allgemein gültigen Vorgehensweisen, vielmehr muss nach heutigem Kenntnisstand jedes Problem, insbesondere jede Parameterkombination, gesondert mit eigenen Methoden untersucht werden.

Das Umgekehrte, nämlich eine obere Schranke für μ oder eine untere Schranke für H anzugeben, löst man in der Regel dadurch, dass man eine konkrete Lösung angibt und deren Werte ermittelt.

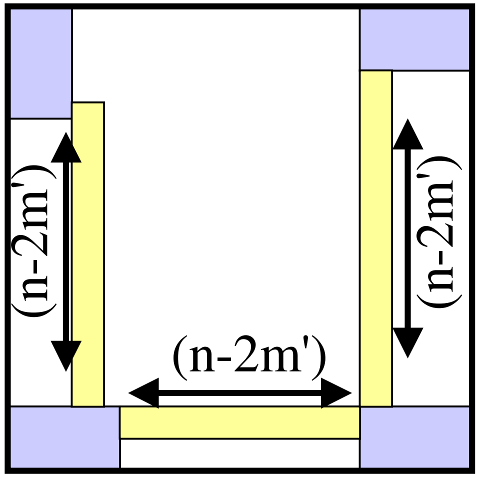
Hinweis zu Beweisen (zehn Folien lang)

Es gibt natürlich die "triviale" obere Schranke H $\leq$ G/m, da sich die Häuser nicht überschneiden dürfen. Dies kann man leicht erweitern zu H $\leq$ (G - 3·(n-2m')) / m, da es Wege zu den Häusern geben muss, die in den vier äußeren Ecken der Gesamtfläche liegen; für diese Wege müssen mindestens 3·(n-2m') Felder verwendet werden; m' ist hierbei die größte Breite bzw. Höhe eines Musters für die Häuser. Skizze hierzu:

Gesamtfläche

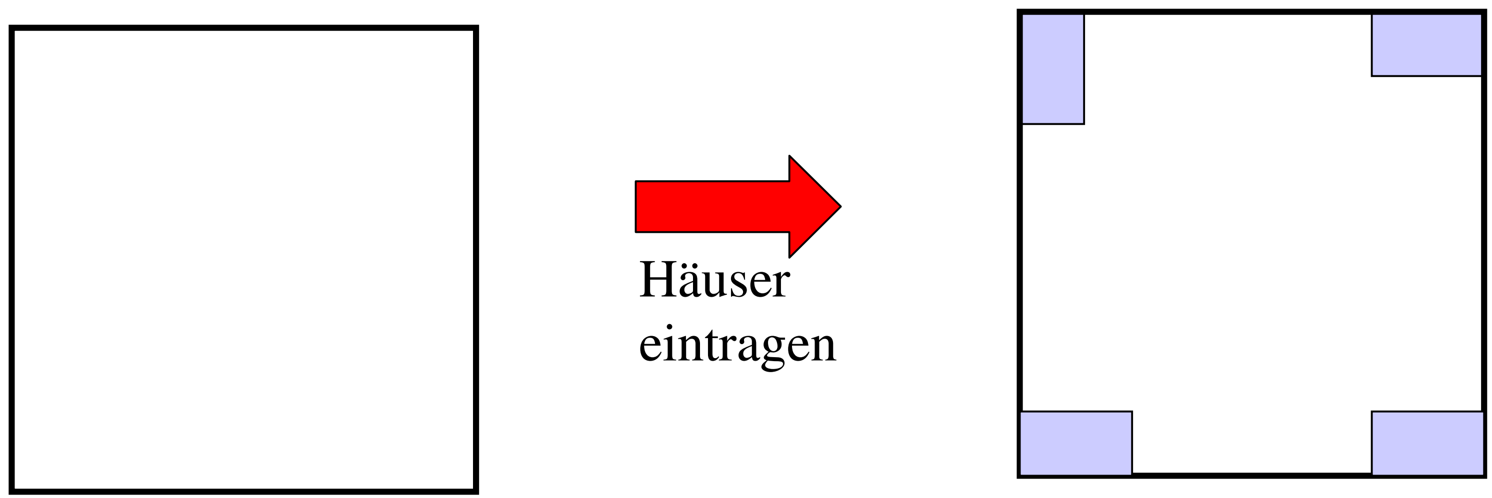
Häuser in den Ecken

Wege dorthin



Für großes n und kleines m' kann man dieses Argument auf verbleibende Felder erneut anwenden und die Abschätzung verschärfen.

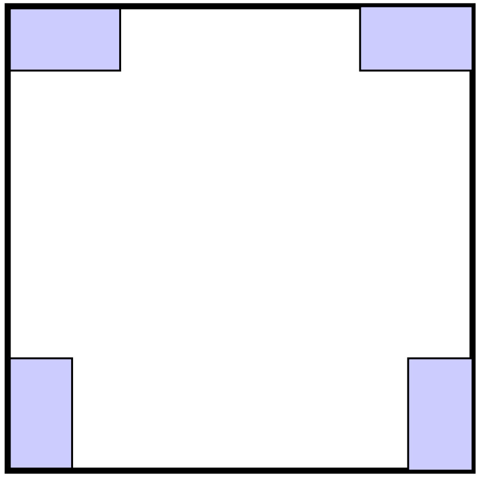
Wir erläutern die Skizze nun genauer. Für eine optimale Lösung, die alle Felder nutzt, müssen an den äußeren Ecken ebenfalls Häuser liegen. Wir tragen diese äußeren Häuser in die Gesamtfläche ein:



Gesamtfläche

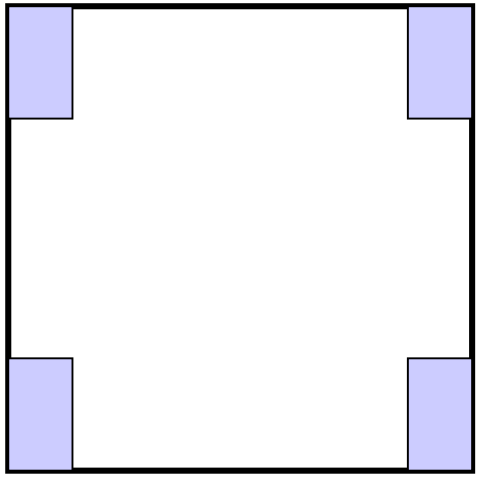
Gesamtfläche mit den vier äußersten Häusern

Die Häuser können natürlich auch anders liegen, das hängt von den vorgegebenen Mustern ab, z.B.:



Gesamtfläche mit den vier äußersten Häusern

oder

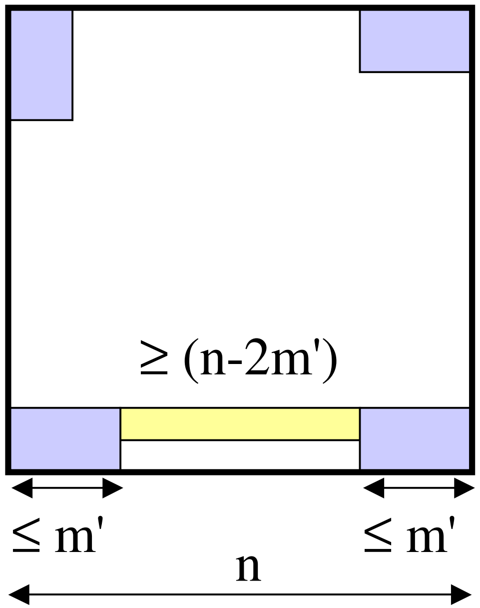


Gesamtfläche mit den vier äußersten Häusern

Diese unterschiedliche Lage berücksichtigen wir dadurch, dass die Zahl m' das *Maximum* von Höhe und Breite ist; dadurch können wir annehmen, dass jedes Haus optimal nahe in Richtung der Wege ausgerichtet ist.

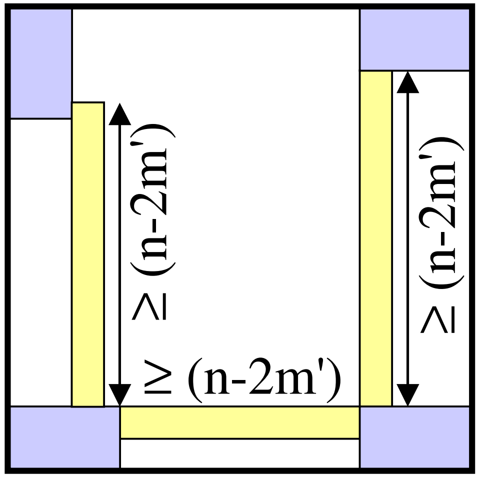
Wir nehmen irgendeine dieser Skizzen.

Da alle Häuser mit dem Startfeld S verbunden sind, sind sie auch untereinander verbunden. Es muss also einen (gelb gezeichneten) Weg zwischen den beiden unteren Häusern geben:



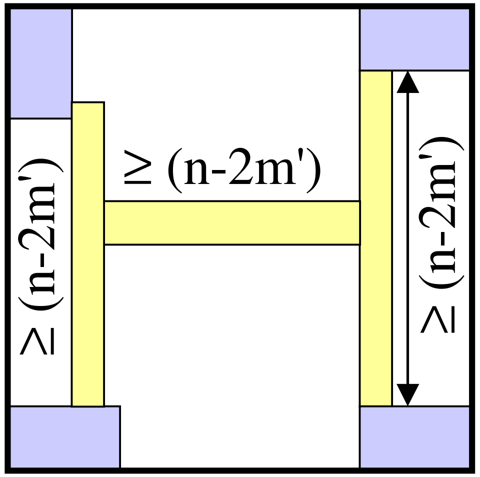
Die Häuser sind höchstens m' Felder von Rand entfernt, der Rand besteht aus n Feldern, also hat der gelbe Weg mindestens die Länge (n-2m').

Das Gleiche gilt nun für die Verbindungen von den unteren zu den oberen Häusern. Die kürzeste Verbindung ist wieder der gerade Weg:



Hier müssen mindestens 3(n-2m') Felder für diese Wege verwendet werden, stehen also nicht für Häuser zur Verfügung.

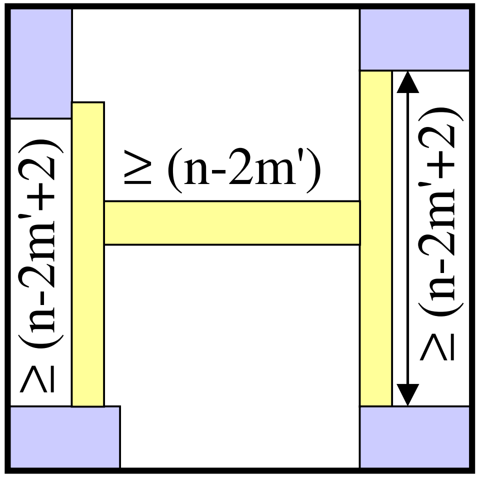
Andere Konstruktionen führen zur gleichen Abschätzung, z.B.:



Keine Konstruktion kann noch schlechter abschneiden. Folglich müssen mindestens 3(n-2m') Felder für diese Wege verwendet werden, stehen also nicht für Häuser zur Verfügung.

Hilfssatz: Die Anzahl H der Häuser beträgt höchstens $H \leq (G - 3\cdot(n-2m')) / m$.

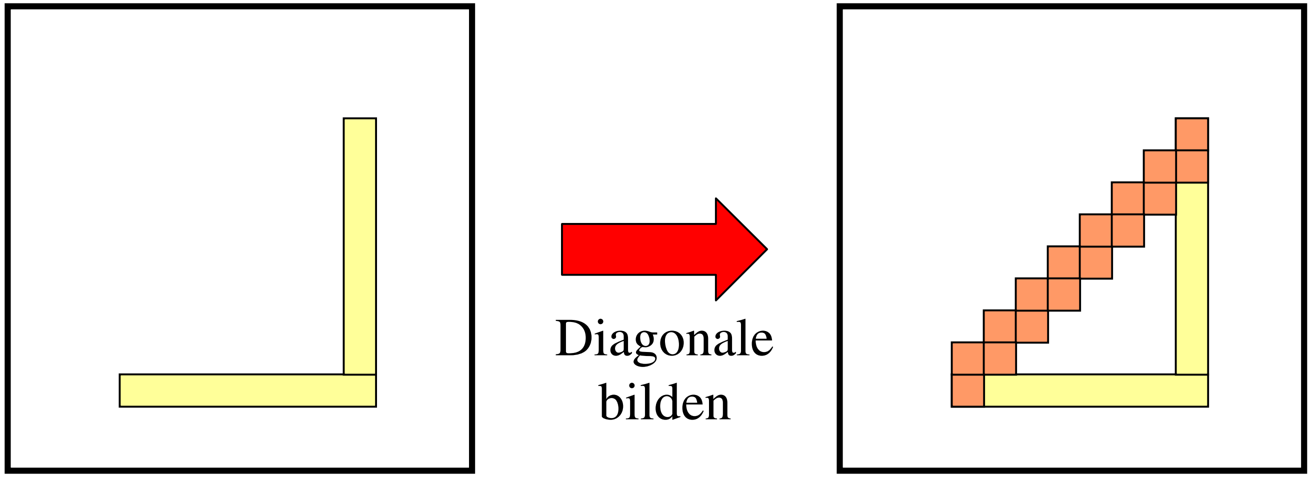
Der Hilfssatz gilt bei quadratischen Mustern der Häuser. In <u>unserem</u> Beispiel sieht man, dass mindestens weitere 2 Felder benötigt werden, da die Muster nicht quadratisch, sondern rechteckig sind, z.B.:



Hilfssatz: Die Anzahl H der Häuser beträgt im Falle unserer 2×3- und 3×2- Muster höchstens

$$H \leq (G - 3\cdot(n-2m') - 2) / m.$$

Dieser Hilfssatz gilt auch noch für Grenzfälle der Form $n < 2\ m'$ (selbst nachprüfen). Ergänzender Hinweis: Man beachte, dass "diagonal verlaufende" Wege nach unserer Definition genau so lang sind wie die Wege auf dem Rand eines Rechtecks:



Daher können wir in dem obigen Beweis von rechteckig angelegten Wegen ausgehen.

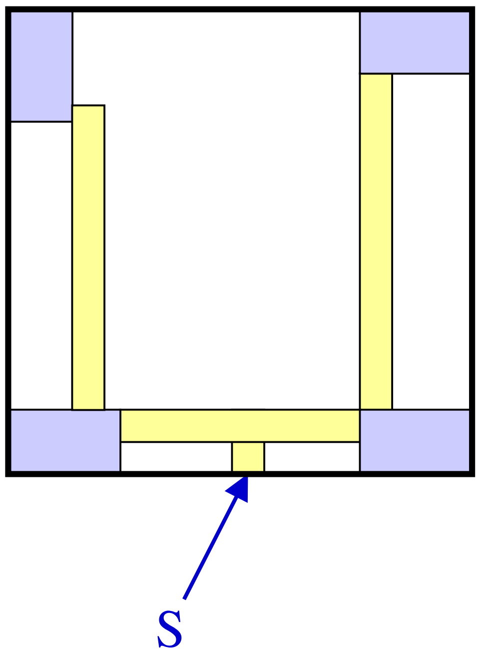
Mit diesem Argument wird auch klar, dass wir uns nicht auf die Häuser *in den Ecken* des Gesamtgrundstücks beschränken müssen, vielmehr genügt es, Häuser zu betrachten, die an den Rändern liegen. Gäbe es solche Häuser nämlich nicht in einer optimalen Lösung, so könnte man zu einem kleineren Grundstück übergehen und die gleiche Betrachtung dort anstellen.

Machen Sie sich dies an Beispielen klar. Beachten Sie hierbei: m' ist die größte Breite bzw. Höhe eines Musters für die Häuser.

Folgerung: Für unser obiges Beispiel ($n=12$, $m=6$, $m'=3$) heißt dies: $H \leq (G - 3\cdot(n-2m')-2) / m = (144-3\cdot(12-2\cdot3)-2)/6 = 124/6$, d.h. $H \leq 20$, da H ganzzahlig sein muss.

Wir haben allerdings nur Lösungen mit höchstens 19 Häusern finden können. Wir vermuten, dass es für dieses spezielle *Beispiel mit n=12 keine Lösung mit 20 Häusern gibt.*

Übrigens: Liegt das Startfeld S am Rande des Gesamtgrund-stücks, so kommen auf jeden Fall noch m'-1 Felder für die Wege hinzu. Beispielskizze hierzu:



Dann gilt also: $H \leq (G - 3 \cdot (n-2m') - m' - 1) / m$

3.) Wir haben das Problem nicht in Beziehung zu anderen Problemen gestellt. Wenn ein Problem besonders hartnäckig ist, so versucht man, es auf ein anderes Problem zurückzuführen, um sich eine Vorstellung vom Schwierigkeitsgrad machen zu können. Das vorgestellte Problem, eine Fläche optimal mit vorgegebenen Mustern auszufüllen, gehört zur Klasse der so genannten *NP-harten Probleme*, von denen man glaubt, dass sie nicht mit effizienten Verfahren (genauer: nicht mit Verfahren, deren Laufzeit polynomiell in der Länge der Eingabewerte ist) gelöst werden können.

Im folgenden Abschnitt werden wir das Füllproblem, also das Problem des optimalen Ausfüllens einer Fläche mit Mustern, exakt formulieren.

5.2 Das Problem exakt formulieren

Will man zu einem Problem eine Lösung finden, die in jedem konkreten Fall von einer Rechenanlage ermittelt werden kann, dann müssen alle Teile des Problems bis ins Kleinste hinein durchdacht und definiert sein.

Wir erstellen also ein (mathematisches/informatisches) **formales Modell** des Problems und legen peinlich genau fest, welche Bedingungen für die einzelnen Bestandteile gelten müssen. Eine Lösung ist dann eine spezielle Ausprägung des Modells (d.h., die variablen Teile sind mit konkreten Mengen und Elementen auszufüllen, deren Beziehungen untereinander den festgelegten Bedingungen genügen). Unter allen Lösungen suchen wir gewisse Lösungen, die zusätzliche Kriterien erfüllen; meist ist eine Funktion zu optimieren. Im Beispiel von Abschnitt 5.1 waren dies der Wert H und anschließend der mittlere Abstand μ.

Für unser Optimierungsproblem ist das formale Modell die Menge von Zahlenpaaren (anschaulich: das ganzzahlige Gitter) mit Zahlenwerten zwischen 1 und einem vorgegebenen Parameter n, der die Größe der quadratischen Grundfläche bestimmt. Diese Menge sollen wir in einzelne Flächen zerlegen, und zwar in solche Flächen, die durch zulässige Muster vorgegeben sind (Bauplätze oder Häuser genannt), und solche, die übrig bleiben und alle Häuser in zu definierender Weise miteinander verbinden (Wege). Hier gibt es eine gewaltige Anzahl von Lösungen, unter denen uns nur die interessieren, die möglichst viele Häuser besitzen und in denen die mittlere Entfernung (mittlere Wegelänge, mittlere Zeitdauer μ) zu einem ausgezeichneten Feld S minimal ist.

Dieser Sachverhalt wird im Folgenden exakt definiert. Hierbei gehen wir in zwei Schritten vor:

- Zuerst beschreiben wir die einzelnen Teile umgangssprachlich und präzisieren einige Bedingungen bereits formal,
- anschließend formulieren wir die Definitionen genau aus.

Es ist ein Lernziel dieser Kurseinheit, dass Sie diesen Prozess selbst durchführen (können).

Zum Vorgehen: Es geht um die Umsetzung von unscharfen Vorstellungen in ein formales Modell. Dieser Prozess läuft nicht so übersichtlich ab, wie man dies üblicherweise in Lehrmaterialien darstellt; vielmehr listet man zunächst viele erforderliche Bestandteile auf, gruppiert diese dann entsprechend ihrer Abhängigkeiten, formalisiert dort, wo es bereits möglich ist, entdeckt in der Regel Lücken, füllt diese durch neue Begriffe und Formalisierungen auf usw. Es ist ein iterativer, tastender Prozess, der dann aufhört, wenn man glaubt, nichts mehr vergessen zu haben, und das Modell alles Wünschenswerte abdeckt. Natürlich enthält das Ergebnis noch Fehler; manche (evtl. alle) Fehler und Mängel bemerkt man später beim Aufstellen der Definitionen.

Dieser Prozess „von der verschwommenen Idee zum formalen Modell“ ist von zentraler Bedeutung, wann immer Aufgaben einer Rechenanlage übertragen werden sollen. Üben Sie diesen Prozess unbedingt ein, indem Sie nach der nächsten Folie das Lernmaterial zur Seite legen und versuchen, ein Modell zur exakten Beschreibung der Aufgabe „Einen Bauplatz optimal auffüllen auf der Basis einer quadratischen Gesamtfläche“ auszuformulieren!

Aufgabe: Entwickeln Sie ein formales Modell zur exakten Beschreibung des Problems „Einen Bauplatz optimal füllen auf der Basis einer quadratischen Gesamtfläche“.

Hinweis zum Vorgehen: Legen Sie dieses Lehrmaterial zur Seite.

a) Schreiben Sie zunächst die erforderlichen Begriffe und Parameter auf, z.B.: n, Weg, Haus, Muster, m, minimale mittlere Entfernung μ usw.
b) Präzisieren Sie dann diese Begriffe umgangssprachlich, wobei Sie sie zugleich in eine Reihenfolge bringen, so dass alle Begriffe, die Sie für die nächste Definition benötigen, bereits definiert sind. Notieren Sie dort, wo es möglich ist, auch schon einige logische Formeln.
c) Definieren Sie dann alle Bestandteile exakt. Prüfen Sie, ob die Definitionen genau Ihrer Vorstellung entsprechen oder ob Fälle erfasst sind, die nicht hinzu gehören, bzw. ob mögliche Ausprägungen durch Ihre Definition nicht beschrieben werden.
d) Brechen Sie nach 40 Minuten ab, kehren Sie zum Lehrmaterial zurück und vergleichen Sie Ihre Ergebnisse mit dessen Definitionen.

Bearbeitungszeit: **40 Minuten**.

40 Minuten später:

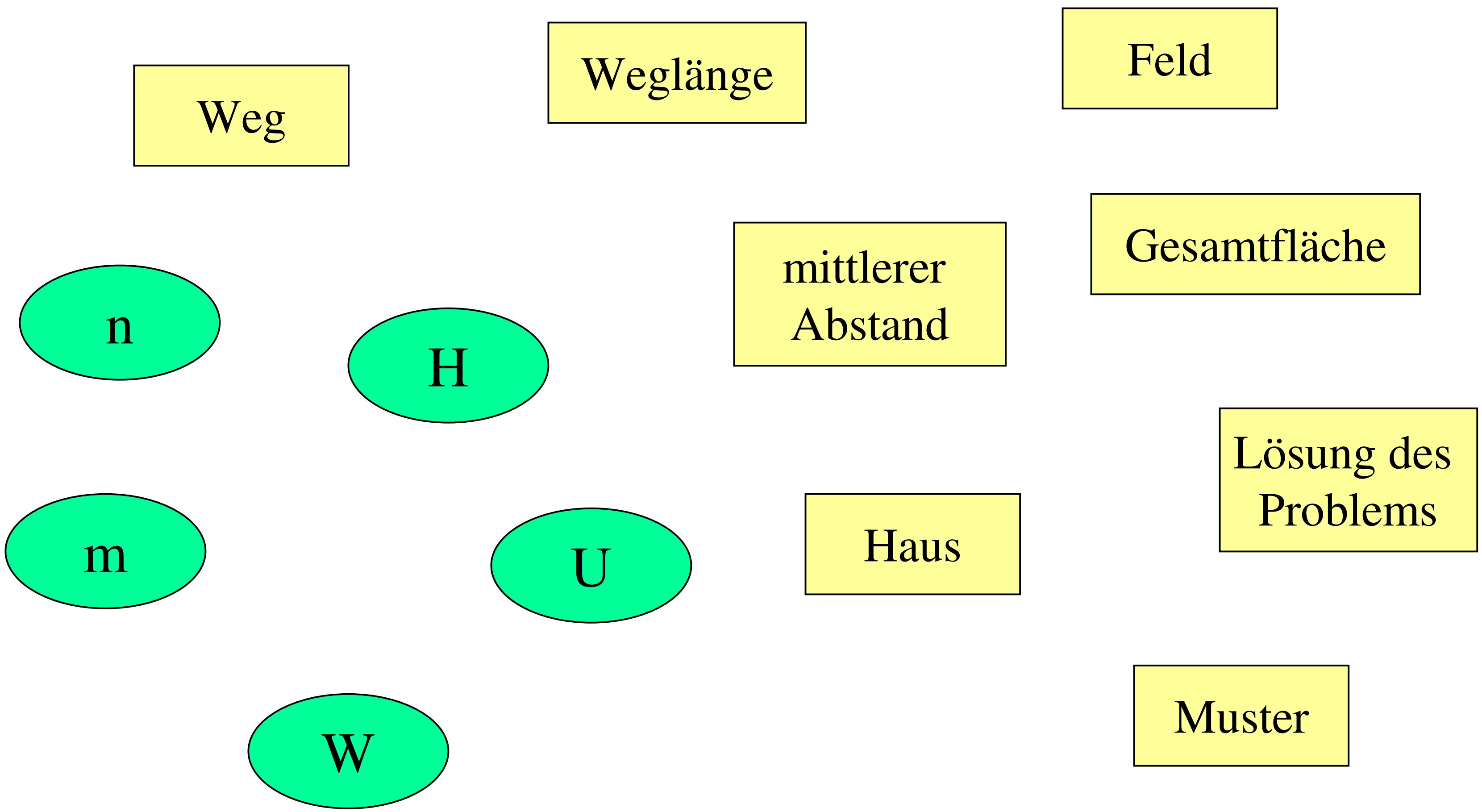
Wir hoffen, Sie konnten einige Präzisierungen treffen, Ordnung in die Begriffswelt bringen und vor allem das Optimierungsziel genau ausformulieren.

Es gibt meist viele Möglichkeiten, ein Modell festzulegen. Im Folgenden geben wir eines durch die entsprechenden Definitionen an. Andere Formalisierungen sind denkbar. Vergleichen Sie daher Ihre Ansätze mit unseren Präzisierungen und versuchen Sie auch zu bewerten, ob Ihre Ansätze möglicherweise besser geeignet sind, das Problem zu beschreiben, als unsere.

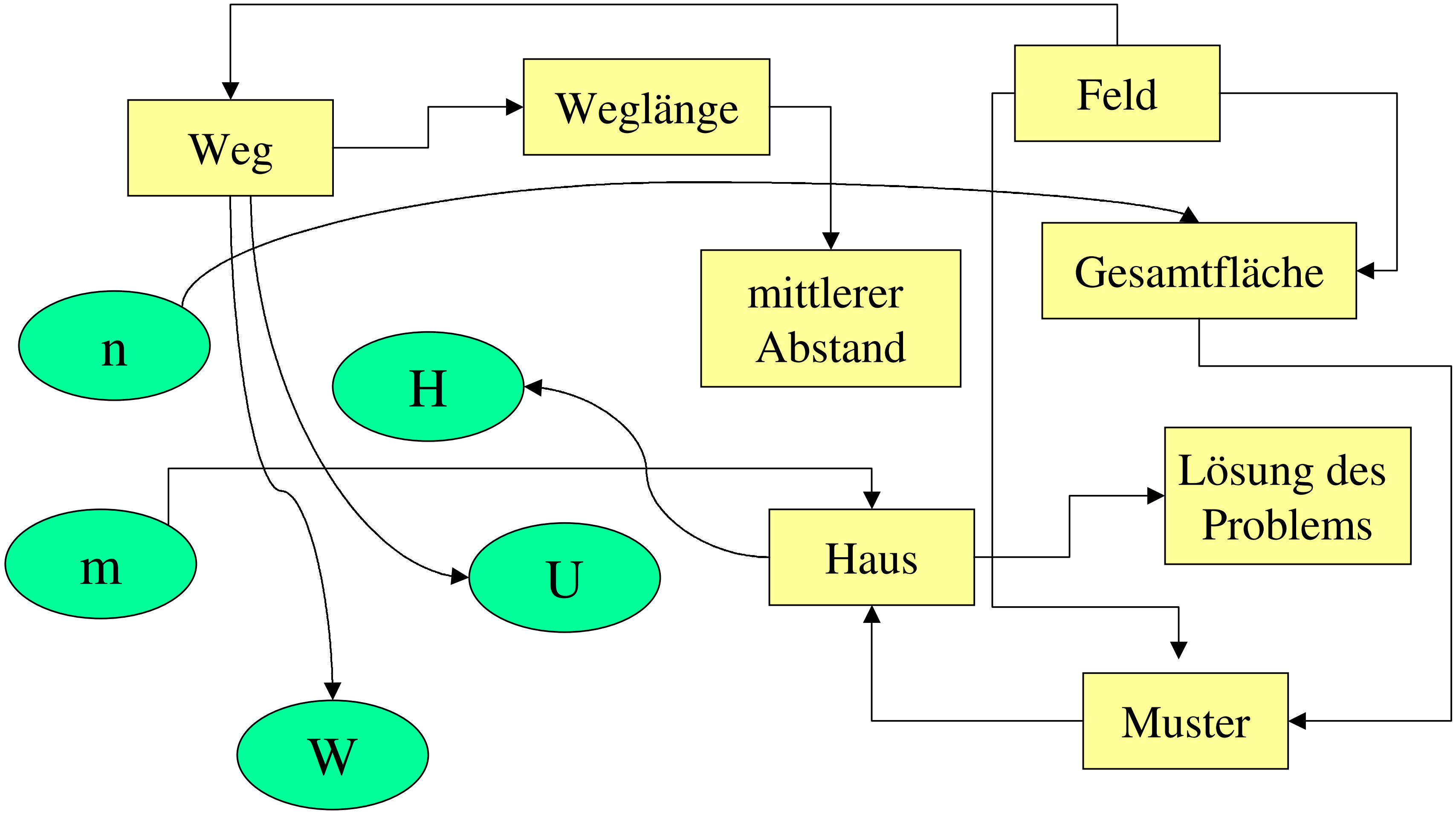
Schritt a:

Wir schreiben zunächst die erforderlichen Begriffe und Parameter auf, z.B.: n, Weg, Haus, Muster, m, minimaler mittlerer Abstand μ usw.
Zugleich versuchen wir bereits, sie so in eine Abhängigkeitsreihenfolge zu bringen, dass alle Begriffe, die wir für ihre Definition benötigen, bereits vorhanden sind.

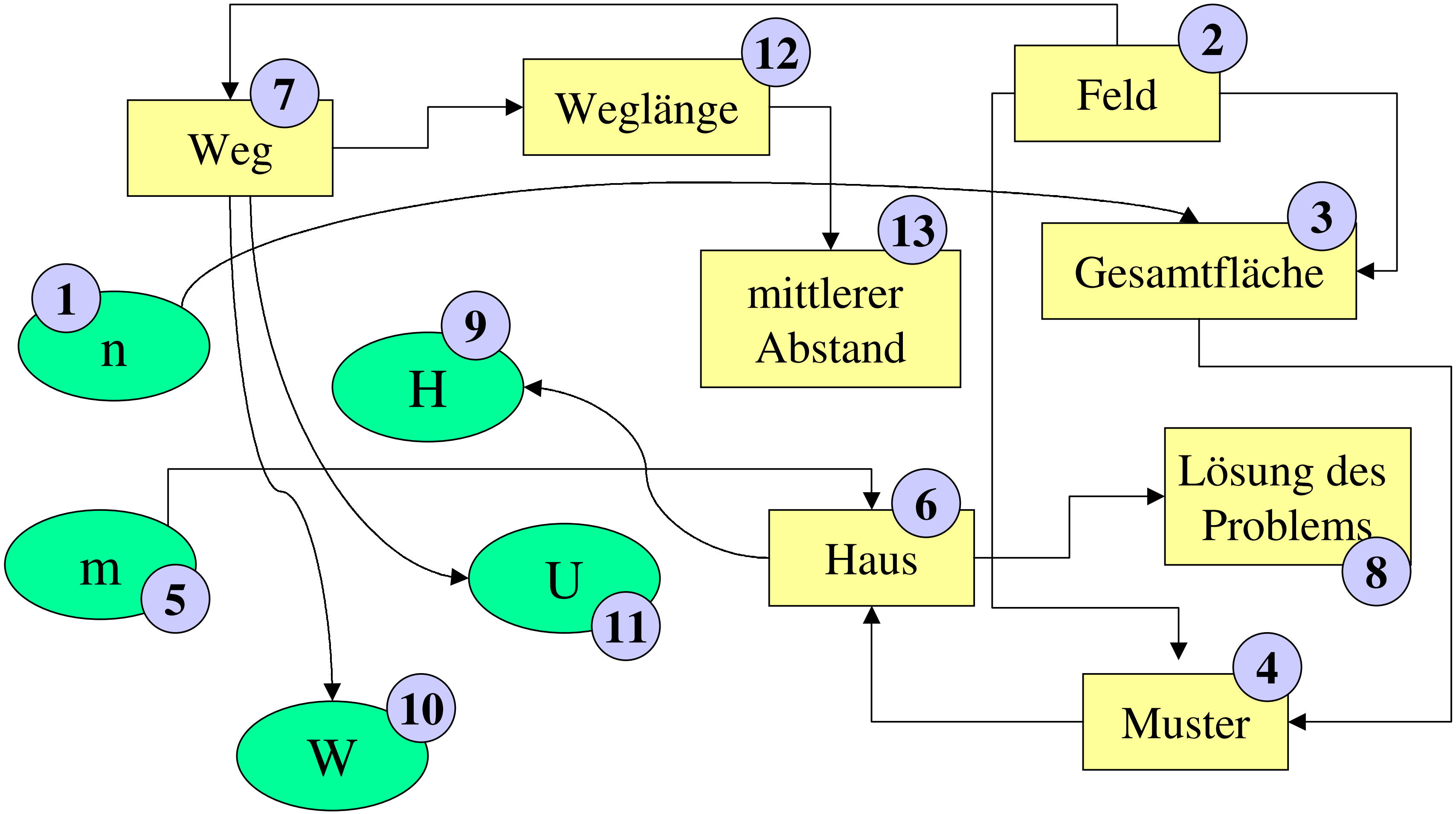
Wir gehen nun so vor, wie es oben beschrieben wurde. Zunächst schreiben wir die im Beispiel von Abschnitt 5.1 verwendeten Begriffe (eckig umrahmt) und Parameter (oval umrahmt) so auf, wie sie uns gerade einfallen. Ein Ergebnis könnte sein:



Danach bringen wir diese Größen in eine logische Abfolge, ausgedrückt durch Pfeile. Z.B.: Um die Gesamtfläche zu beschreiben, braucht man zunächst die Seitenlänge n; um ein Haus zu definieren, muss erst das Muster bekannt sein, usw. Das Ergebnis könnte sein:



Jetzt legen wir willkürlich eine Reihenfolge fest, in der diese Begriffe definiert werden sollen und die diesen Pfeilen nicht widerspricht, z.B.



Schritt b:

Wir präzisieren diese Begriffe umgangssprachlich, wobei wir sie zugleich in eine Reihenfolge bringen, die den Abhängigkeiten aus Schritt a entspricht. Dort, wo es möglich ist, notieren wir auch schon einige logische Formeln, vor allem bei notwendigen Bedingungen, die die Größen erfüllen müssen. Hilfsbegriffe und eventuell "vergessene" Begriffe fügen wir hinzu.

Unter Schritt a) nicht betrachtete Begriffe sind S, Z und "optimale Lösung".

Auflistung der erforderlichen Begriffe (1):

n Wir gehen von einer quadratischen Fläche der Seitenlänge n aus.

Feld: Jedes Feld wird durch sein Tupel (i,j) mit $1 \leq i \leq n$ und $1 \leq j \leq n$ beschrieben.

S ist das Startfeld; genauer: S ist das Feld, auf das man im ersten Schritt gehen muss.

G ist die Gesamtfläche ausgedrückt als Anzahl aller Felder, d.h. $G = n \times n$ (Felder).

Muster: Ein Muster ist eine Menge von Feldern. Wir betrachten hier nur „rechteckig zusammenhängende Muster“, d.h., wir fordern zusätzlich: Besteht das Muster aus mindestens zwei Feldern, so muss es zu jedem Feld ein weiteres Feld in dieser Menge geben, so dass diese beiden Felder rechteckig aneinander stoßen. Formal: Für jedes Muster $z = \{(i_1,j_1), (i_2,j_2), ..., (i_k,j_k)\}$ muss gelten:

$1 \leq i_r \leq n$ und $1 \leq j_r \leq n$ für $r = 1,..., k$,

und wenn $k > 1$ ist, dann muss es zu jedem $(i,j) \in z$ ein $(i',j') \in z$ geben mit

entweder $|i-i'| = 1$ oder $|j-j'| = 1$

(Erfahrene merken sogleich: Dies ist nicht korrekt formalisiert. Warum?)

m ist Zahl der Felder für ein Haus. (Für ein Problem geben wir noch eine Menge Z von Mustern aus m Feldern vor, die für Häuser verwendet werden dürfen.)

Auflistung der erforderlichen Begriffe (2):

Z ist die Menge der „zulässigen Muster", d.h., Z hat die Form
$Z = \{z_1, z_2, ..., z_s\}$ und jedes Muster $z_r \in Z$ besteht aus genau m Tupeln.

Haus: Ein Haus h ist eine Teilfläche der Gesamtfläche bestehend aus m Feldern, die einem Muster aus Z entspricht. Dies lässt sich formal so darstellen, dass ein Haus bis auf ein additives Tupel (x,y) gleich einem zulässigen Muster aus Z ist, d.h.,
$h = \{(x_1,y_1), (x_2,y_2), ..., (x_m,y_m)\}$ mit $1 \leq x_r \leq n$ und $1 \leq y_r \leq n$ für $r = 1,..., m$
und es gibt ein $z \in Z$, $z = \{(i_1,j_1), (i_2,j_2), ..., (i_m,j_m)\}$ und ein Tupel (x,y) von natürlichen Zahlen mit $x_r = i_r + x$ und $y_r = j_r + y$ für $r = 1,..., m$.

Weg: Ein Weg ist eine Folge von Feldern, bei der aufeinander folgende Felder stets rechteckig aneinander stoßen und die keine Felder enthält, die von einem Haus überdeckt werden.

Lösung: Eine Lösung ist eine überlappungsfreie Platzierung von Häusern auf der Gesamtfläche, so dass jedes Haus über einen Weg von S aus erreicht werden kann; genauer: Für jedes Haus h gibt mindestens einen mit S beginnenden Weg, dessen letztes Feld rechteckig an ein Feld des Hauses h stößt.

Auflistung der erforderlichen Begriffe (3):

H = Zahl der bei der jeweiligen Lösung platzierten Häuser.
W = Zahl der Felder, die bei der jeweiligen Lösung für Wege dienen.
U = Zahl der bei der jeweiligen Lösung von **S** aus unerreichbaren Felder.
Hinweis: Zur Kontrolle: Es muss stets gelten: $\mathbf{G = H \times m + W + U}$.

Weglänge: Der Abstand **w(h)** eines Hauses h ist die kürzeste Anzahl an Feldern eines Weges, der von S zu einem Feld führt, an das h rechteckig (also nicht diagonal) angrenzt. (Das Feld S zählt hierbei mit.)

μ ist für eine Lösung der mittlere Abstand zu einem Haus, d.h.:
Wenn $h_1, h_2, \ldots, h_H$ alle Häuser der Lösung mit Abständen $w(h_i)$ sind, dann ist
$\mu = (w(h_1) + w(h_2) + \ldots + w(h_H)) / H$.

Optimale Lösung: Eine Lösung heißt optimal, wenn es keine Lösung mit einem größeren Wert H gibt und wenn es unter allen Lösungen mit maximalem H keine Lösung mit einem kleineren μ gibt. (Es wird also auf jeden Fall eine maximale Zahl an Häusern angestrebt und erst dann wird der mittlere Abstand minimiert.)

Dieser Schritt b) gilt als der heikelste bei der Modellbildung, da er die „Architektur des Modells“ festlegt.

Was man hier "falsch" macht, lässt sich später nur noch schwer korrigieren. Dabei sind es meist keine inhaltlichen Fehler, die man begeht, sondern nur Entscheidungen, die künftige Fragen nicht richtig einbeziehen und die sich daher später als nachteilig herausstellen.

Schritt c:

Wir definieren nun alle Bestandteile exakt. Wir prüfen, ob die Definitionen genau unseren Vorstellungen entsprechen.

Die nun folgende Umsetzung der Ergebnisse aus Schritt b) in einen durchgängigen Formalismus entspricht der Codierung/Implementierung in einer Programmiersprache, die mit einiger Erfahrung relativ leicht bewerkstelligt werden kann. Hier fallen noch wichtige Entscheidungen, vor allem die Wahl einer geeigneten Datenstruktur. Meist entdeckt man in dieser Umsetzung diverse Versäumnisse oder sogar Fehler.

Gegebenenfalls muss auch das Modell entsprechend verbessert werden. Dies alles trifft jedoch hier nicht zu, da unser Problem recht gut überschaubar ist. Bei sehr komplexen Modellen muss dagegen eine häufigere Rückkopplung und Inspektion mit den unter Schritt a) und b) festgelegten Vorstellungen erfolgen.

Definition 5.1:

Es sei n eine natürliche Zahl, $n \geq 1$.

a) Ein <u>Feld</u> ist ein Tupel (i,j) mit $1 \leq i \leq n$ und $1 \leq j \leq n$.

b) Ein <u>Weg</u> von einem Feld (i_1,j_1) zu einem Feld (i_k,j_k) ist eine Folge von Feldern $(i_1,j_1), (i_2,j_2), \ldots, (i_k,j_k)$ mit: $k \geq 1$ und

für alle $r = 1, \ldots, k-1$ gilt

entweder $|i_r - i_{r+1}| = 1$ und $j_r = j_{r+1}$

oder $|j_r - j_{r+1}| = 1$ und $i_r = i_{r+1}$.

(Dies besagt, dass das Feld (i_r,j_r) waagerecht oder senkrecht neben dem Feld (i_{r+1},j_{r+1}) liegt für alle $r = 1, \ldots, k-1$.)

c) Die <u>Länge eines Weges</u> $(i_1,j_1), (i_2,j_2), \ldots, (i_k,j_k)$ ist $k-1$.

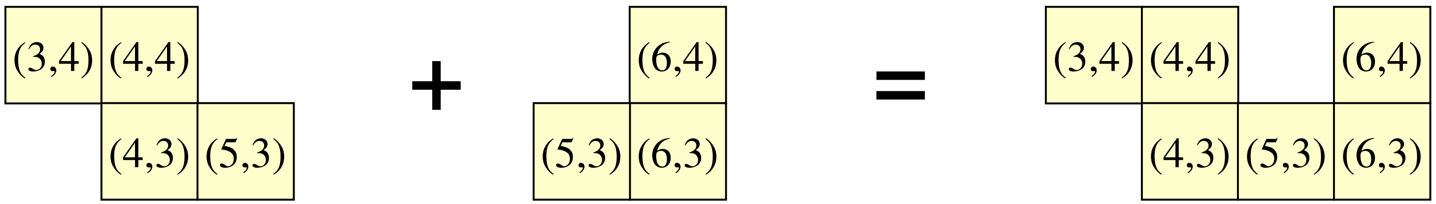
d) Ein Weg $(i_1,j_1), (i_2,j_2), \ldots, (i_k,j_k)$ heißt <u>einfach</u>, wenn alle Felder des Weges paarweise verschieden sind.

e) Ein Weg $(i_1,j_1), \ldots, (i_k,j_k)$ heißt <u>Zyklus</u>, wenn $(i_1,j_1)=(i_k,j_k)$, $k \geq 3$ und $(i_1,j_1), (i_2,j_2), \ldots, (i_{k-1},j_{k-1})$ ein einfacher Weg sind.

Überprüfung der Definition 5.1:

Wie üblich definieren wir die Länge eines Weges als die Zahl der Schritte, um alle Felder des Weges zu durchlaufen. Ein Weg der Länge 0 besteht aus einem Feld.

Intuitiv muss die Länge so definiert werden, dass die Länge zweier Wege, die aneinander gehängt werden können (weil das letzte Feld des einen Weges zugleich das erste Feld des anderen Weges ist), gleich der Summe der Weglängen der Einzelwege ist. Dies trifft zu. Beispiel hierfür:



Länge: 3 Länge: 2 Länge: 5

Für den Abstand vom Startfeld S zu einem Haus werden wir aber diese Länge um eins erhöhen müssen, da wir annehmen, dass man beim Betreten der Gesamtfläche den ersten Schritt von außen auf das Feld S macht. Auf diese Besonderheit müssen wir später achten.

Definition 5.2:

Es sei n eine natürliche Zahl, $n \geq 1$.

a) Ein <u>Muster</u> z ist eine nicht-leere Menge von Feldern
$z = \{(i_1,j_1), (i_2,j_2), \ldots, (i_m,j_m) \mid m \geq 1,\ 1 \leq i_r \leq n \text{ und } 1 \leq j_r \leq n \text{ für } r = 1,\ldots, m\}$
mit folgender Zusatzbedingung für $m>1$:

für alle $(i,j), (i',j') \in z$ gibt es einen Weg von (i,j) nach (i',j'), der nur über Felder verläuft, die in z liegen, d.h.:

$$\boxed{\begin{array}{l} \forall (i,j),(i',j') \in z \text{ mit } (i,j) \neq (i',j') \\ \exists \text{ Weg } (i_1,j_1), (i_2,j_2),\ldots, (i_k,j_k) \text{ mit } (i,j) = (i_1,j_1), (i',j') = (i_k,j_k) \\ \quad \text{und } \forall r = 1,\ldots, k: (i_r,j_r) \in z \end{array}}$$

b) $m = |z|$ heißt die <u>Größe des Musters</u> z.

c) Eine Menge $Z = \{z_1, z_2, \ldots, z_s\}$ heißt <u>Menge zulässiger Muster</u>, wenn jedes z_r ein Muster gemäß Definition 5.2 a) ist und wenn alle Muster z_r die gleiche Größe besitzen.

Überprüfung der Definition 5.2:

Das Muster soll "rechteckig zusammenhängend" sein; gemeint ist hiermit, dass man von jedem Feld des Musters zu jedem anderen Feld durch waagerechte und senkrechte Schritte innerhalb des Musters gelangen kann, d.h., dass es von jedem Feld zu jedem anderen Feld einen Weg im Muster gibt. Also: Für je zwei verschiedene Felder (i,j) und (i',j') des Musters gibt es einen Weg von (i,j) nach (i',j'), der nur aus Feldern des Musters besteht. Genau dies besagt die Formel:

$$\forall (i,j),(i',j') \in z \text{ mit } (i,j) \neq (i',j')$$
$$\exists \text{ Weg } (i_1,j_1), (i_2,j_2),\ldots, (i_k,j_k) \text{ mit } (i,j) = (i_1,j_1), (i',j') = (i_k,j_k)$$
$$\text{und } \forall r = 1,\ldots, k\text{: } (i_r,j_r) \in z$$

m war in unseren Beispielen bereits der Parameter für die Größe des Musters.

Vereinfachend hatten wir angenommen, dass alle zulässigen Muster für Häuser die gleiche Größe haben sollen. Diese Bedingung kann man auch fallen lassen.

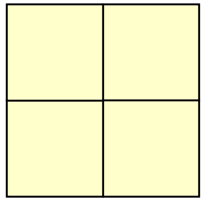
<u>Beispiele</u>. Muster sind:

$z_1 = \{(5,3), (6,3), (5,4), (6,4)\}$,
Größe: 4.

Veranschaulichung:

(5,4)	(6,4)
(5,3)	(6,3)

Genügen würde bereits folgende Darstellung:

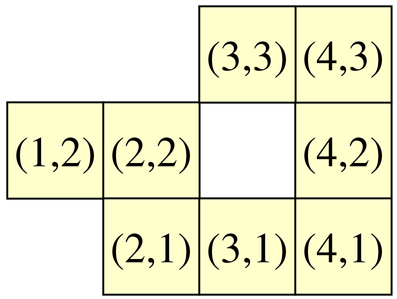


d.h., das Muster z_1 und folgendes Muster
$z_2 = \{(1,1), (2,1), (1,2), (2,2)\}$ sind für unsere Zwecke gleich, auch wenn sie aus formaler Sicht verschiedene Muster sind.

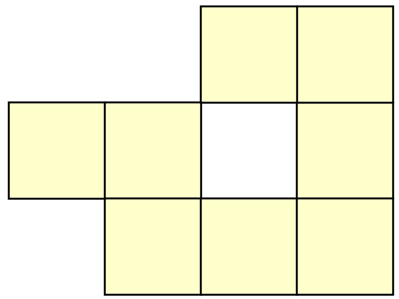
Ein weiteres Muster ist:

$z_3 = \{(4,2), (2,1), (1,2), (3,3), (4,3), (4,1), (2,2), (3,3)\}$,
Größe: 8.

Veranschaulichung:

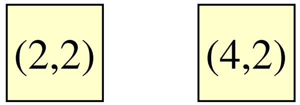


Genügen würde bereits folgende Darstellung:

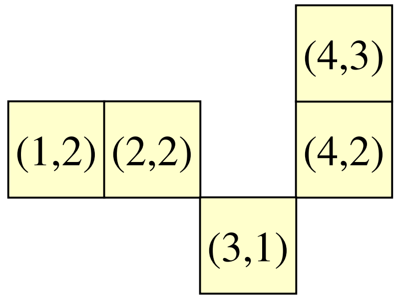


Keine Muster dagegen sind:

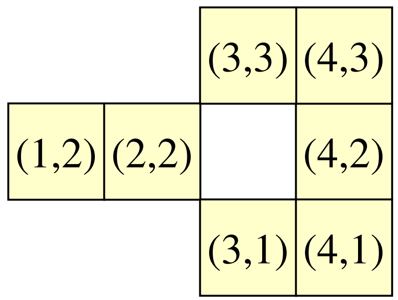
{(2,2), (4,2)}, Veranschaulichung:



{(1,2), (2,2), (3,1), (4,3), (4,2)}



{(1,2), (2,2), (3,1), (3,3), (4,1), (4,2), (4,3)}



Diese Überprüfung zeigt:

Vom Begriff "Muster" benötigen wir eigentlich nur die relative Anordnung von Einheitsquadraten zueinander. Dies könnte man erreichen, indem man die Nummerierung der Felder normiert, also stets das Minimum der x- und der y-Komponente auf 1 setzt. Wir wollen dies nur feststellen, ohne die Definition 5.2 neu zu fassen.

[Hinweis:
Aus theoretischer Sicht ist ein Muster eine Äquivalenzklasse aller Muster der Definition 5.2, wobei genau die Muster in der gleichen Äquivalenzklasse zusammenfasst werden, die sich bis auf eine Verschiebung in der Ebene nicht unterscheiden. Wir werden aber solche Überlegungen hier nicht vertiefen.]

Definition 5.3:

Es seien n eine natürliche Zahl, $n \geq 1$, und Z eine Menge zulässiger Muster der Größe m.

a) Ein Haus h bzgl. Z ist eine nicht-leere Menge von Feldern

$h = \{(x_1,y_1), (x_2,y_2), \ldots, (x_m,y_m) \mid m \geq 1,\ 1 \leq x_r \leq n \text{ und } 1 \leq y_r \leq n \text{ für } r=1,\ldots,m\}$

mit folgender Zusatzbedingung für $m>1$:

Es gibt ein Muster $z \in Z$ und ein Paar (x,y) ganzer Zahlen mit $h = z + (x,y)$, d.h.:

$\exists\, z = \{(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)\} \in Z$ und $\exists\, (x,y)$ mit

$h = \{(i_1+x, j_1+y), (i_2+x, j_2+y), \ldots, (i_m+x, j_m+y)\}$.

noch Definition 5.3:

b) Ein Feld (i,j) <u>grenzt an</u> ein Haus $h = \{(x_1,y_1), (x_2,y_2), ..., (x_m,y_m)\}$, wenn es ein r ($1 \le r \le m$) gibt mit

entweder $|x_r-i| = 1$ und $y_r = j$

oder $|y_r-j| = 1$ und $x_r = i$.

c) Eine nicht-leere Menge B von Häusern bzgl. Z

$B=\{h_1, h_2, ..., h_{|B|}\}$

heißt <u>überschneidungsfrei</u>, wenn für alle Häuser $h_q, h_r \in B$ mit $q \neq r$ gilt: $h_q \cap h_r = \emptyset$.

Überprüfung der Definition 5.3:

Ein Haus muss gleich einem Muster sein, bis auf Verschiebung in der Ebene. Dementsprechend muss $h = z + (x,y)$ für ein Muster $z \in Z$ und einen Verschiebevektor (x,y) sein. Da die Muster nicht normiert sind, können in dem Verschiebevektor auch negative Zahlen auftreten. Die Definition 5.3 a) erfüllt also genau die intuitive Vorstellung.

Wir wollen später eine Menge von Häusern finden, die sich gegenseitig nicht überlappen. Dies sind genau überschneidungsfreie Mengen von Häusern aus Definition 5.3 c).

Die Häuser müssen später durch Wege mit dem Startfeld S verbunden sein. Hierzu wird festgelegt, welche Felder an ein Haus grenzen. Dies sind genau diejenigen aus Definition 5.3 c), die waagerecht oder senkrecht an ein Feld des Hauses stoßen.

Definition 5.4:

Es seien n eine natürliche Zahl, $n \geq 1$, Z eine Menge zulässiger Muster der Größe m, B eine überschneidungsfreie Menge von Häusern bzgl. Z sowie S ein Feld.

a) Der <u>Abstand</u> $w_{h,S}$ <u>eines Hauses h zu S</u> ist die Länge eines kürzesten Weges (plus 1), der von S zu einem Feld führt, das an h grenzt, und der nur über Felder führt, die nicht zu einem Haus aus B gehören, d.h.:

$w_{h,S} = \mathrm{Min}\ \{\ k \mid$ Es gibt ein an h grenzendes Feld (i_k,j_k) und einen Weg $(i_1,j_1), (i_2,j_2), \ldots, (i_k,j_k)$ von $(i_1,j_1) = S$ nach (i_k,j_k) und für alle Häuser $h' \in B$ gilt: Keines der Felder (i_r,j_r) gehört zu h' $(1 \leq r \leq k)$ $\}$

bzw.

$w_{h,S} = \infty$, falls es kein solches angrenzendes Feld oder keinen solchen Weg gibt.

noch Definition 5.4:

b) <u>Der mittlere Abstand</u> von $B=\{h_1, h_2, ..., h_{|B|}\}$ zu S ist

$$\mu_{B,S} = (w_{h_1,S} + w_{h_2,S} + ... + w_{h_{|B|},S}) / |B|.$$

Überprüfung der Definition 5.4:

Wie bereits angekündigt muss der Abstand eines Hauses zum Startfeld S definiert werden. In den Beispielen in Abschnitt 1 haben wir den "Eingang" eines jeden Hauses, dargestellt durch rote Punkte, optimal nahe bzgl. S gelegt. Formal heißt dies: Suche für jedes an das Haus h grenzende Feld den kürzesten Weg zu S, der zu den Verbindungswegen gehört, d.h., der nicht über Häuser aus B führt, und wähle unter allen diesen Feldern dasjenige mit der kürzesten Weglänge zu S. Genau diese Entfernung $w_{h,S}$ wird in Definition 5.4 a berechnet. Man beachte, dass nicht die Weglänge, sondern die um 1 erhöhte Weglänge genommen wird; denn das Minimum erfolgt über k und nicht über die Länge des Weges k-1, vgl. Definition 5.1.

$\mu_{B,S}$ ist der übliche Mittelwert aller Abstände.

Definition 5.5:

Es seien n eine natürliche Zahl, $n \geq 1$, Z eine Menge zulässiger Muster und S ein Feld.

a) (n, Z, S) heißt ein Füllproblem.

b) Eine Menge B von überschneidungsfreien Häusern bzgl. Z heißt Lösung des Füllproblems (n, Z, S), wenn $w_{h,S} \neq \infty$ für alle $h \in B$ gilt.

c) Eine Lösung B des Füllproblems (n, Z, S) heißt optimal, wenn es keine Lösung B' dieses Füllproblems mit $|B'| > |B|$ gibt und wenn für alle Lösungen B" dieses Füllproblems mit $|B| = |B''|$ gilt: $\mu_{B,S} \leq \mu_{B'',S}$.

Überprüfung der Definition 5.5:

Unsere Aufgabe lautete: Gegeben seien eine Seitenlänge n der quadratischen Gesamtfläche, eine Menge von zulässigen Mustern und ein Startfeld S. Definition 5.5 a) besagt, dass durch n, Z und S das Problem vollständig spezifiziert ist.

Fülle dann zu vorgegebenem Startfeld S die Gesamtfläche mit Häusern aus, so dass jedes Haus von S aus erreicht werden kann. Dies drückt Definition 5.5 b) offensichtlich aus.

Schließlich möchte man möglichst viele Häuser unterbringen und zusätzlich den mittleren Abstand $\mu_{B,S}$ minimieren, wie es in Definition 5.5 c) formuliert ist.

Hinweis:

Die Definition 5.5 enthält zwei Typen von Problemen:
Ein **Entscheidungsproblem** und
ein **Optimierungsproblem**.

Das <u>Entscheidungsproblem</u> lautet: Stelle fest, ob eine vorgegebene Zahl von Häusern unter den Nebenbedingungen platziert werden kann. Als Antwort wird prinzipiell nur "Ja" oder "Nein" erwartet, auch wenn man im Falle "Ja" gerne eine konkrete Lösung sehen möchte.

Das Optimierungsproblem lautet: Finde heraus, wie viele Häuser maximal platziert werden können und liefere unter allen solchen Lösungen diejenige mit minimalem mittlerem Abstand.
Hier wird in jedem Fall erwartet, dass es Lösungen gibt und man unter diesen eine optimale aufzuspüren hat.

Das Optimierungsproblem kann nicht leichter zu lösen sein als ein zugehöriges Entscheidungsproblem, da die optimale Lösung zugleich die Entscheidung liefert, ob eine Lösung mit den vorgegebenen Eigenschaften existiert. In vielen Fällen hängen diese beiden Typen von Problemen eng zusammen. Wir gehen hierauf nicht ein, sondern verweisen auf die Literatur über Komplexitätstheorie.

Durch die Definitionen 5.1 bis 5.5 ist jetzt das Modell für das

(2-dimensionale) "Füllproblem"

fertig gestellt. Das Problem ist so weit formalisiert, dass es losgelöst von allen sonstigen Einschränkungen der Praxis untersucht werden kann. Ziel wird es sein,

- einen Algorithmus zur optimalen Lösung des Füllproblems anzugeben,
- einen Algorithmus zur näherungsweisen Lösung zu formulieren, der möglichst schnell arbeitet,
- die Eigenschaften des Problems zu untersuchen und vorab obere/untere Schranken zu berechnen,
- die "Schwierigkeit" des Problems zu ermitteln.

Diese Fragen werden in den kommenden Abschnitten in Angriff genommen. Zuerst betrachten wir einfach aussehende Spezialfälle, an denen die Algorithmen erläutert werden.

5.3 Spezialfall "Rucksackproblem"

Beim Füllproblem versuchten wir, möglichst viele Objekte, die zulässige Muster sein müssen, auf einer gegebenen Fläche unterzubringen, wobei eine Nebenbedingung, nämlich die Erreichbarkeit vom Startfeld aus, erfüllt sein muss.

Wir gehen nun von der zweidimensionalen Fläche zur Eindimensionalität, also zur Menge der natürlichen Zahlen über. "Zulässige Muster" werden dann ebenfalls natürliche Zahlen. Die Nebenbedingung lassen wir weg, erlauben dafür aber eine große Menge an Mustern, was prinzipiell beim Füllproblem ebenfalls möglich war.

So erhalten wir folgendes Problem:

Gegeben eine Zahl G (bisher: die Gesamtfläche) und eine Folge von natürlichen Zahlen (bisher die Menge aller möglichen Häuser). Man stelle für eine Zahl H (bisher: Anzahl der zu platzierenden Häuser) fest, ob es H Zahlen in der Folge gibt, deren Summe gleich G ist (bisher: ob es H Häuser gibt, die exakt die Fläche ausfüllen, allerdings musste noch die Nebenbedingung gelten und es waren unerreichbare Felder erlaubt).

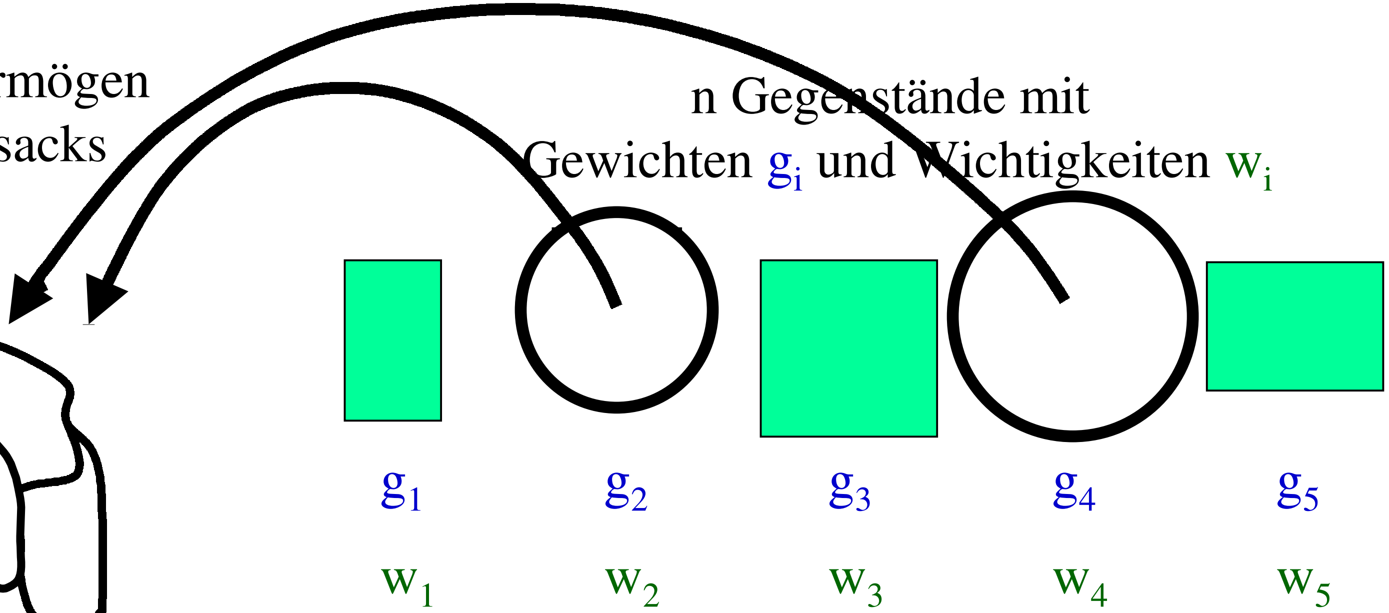
Wir schreiben dies in knapper Form auf:

Gegeben: zwei natürliche Zahlen G und H und eine Folge natürlicher Zahlen $a_1, a_2, ..., a_k$ mit $k > H$. Gesucht: eine H-elementige Teilfolge $a_{i_1}, a_{i_2}, ..., a_{i_H}$, deren Summe gleich G ist [oder die möglichst dicht an G (von unten) herankommt] ?

Dies ist eine Variante des so genannten *Rucksackproblems* (statt k verwenden wir, wie in der Literatur üblich, den Buchstaben n, der gern als Maß für die Größe eines Problems genommen wird):

Gegeben seien n Gegenstände, deren Gewichte $g_1, g_2, ..., g_n$ seien. Der Rucksack hat eine maximale Tragfähigkeit von G Gewichtseinheiten; wird mehr in ihn gefüllt, reißt er und wird unbrauchbar. Jeder der Gegenstände hat zusätzlich einen Wert (oder eine Wichtigkeit oder einen Nutzen), der mit $w_1, w_2, ..., w_n$ beziffert wird. Man soll den Rucksack nun mit Gegenständen so füllen, dass alle eingepackten Gegenstände höchstens das Gewicht G besitzen und dass deren Wichtigkeit mindestens einen vorgegeben Wert W erreicht.

Z.B. die Gegenstände 2 und 4 einpacken:



Es muss gelten:

Gewicht im Rucksack: $g_2 + g_4 \leq G$

Angestrebt wird beim Einpacken:

Wichtigkeit (Nutzen): $w_2 + w_4 \geq W$

Anwendung: Auf eine U-Boot- oder eine Weltraumfahrt darf jede(r) Mitreisende persönliche Gegenstände eigener Wahl mitnehmen, die pro Person ein Gesamtgewicht G nicht übersteigen; jede(r) wird sie so auswählen, dass der erwartete persönliche Nutzen dieser Gegenstände möglichst groß wird bzw. einen vorgegebenen "Nützlichkeitswert" W übersteigt.

Ähnliche Anwendung: Bei der Konzipierung eines Autos kann man viele verschiedene Annehmlichkeiten einbauen, z.B. Klimaanlage, kleine Elektromotoren für alle möglichen Hilfen, einen Wassertank, einen größeren Benzintank, eine kleine Bar, diverse Computer, stabilere Achsen usw. Es passt aber nicht alles raum- oder gewichtsmäßig hinein. Man wird daher (auf Grund von Kundenbefragungen) diesen Geräten eine Wichtigkeit zuordnen und muss dann ermitteln, wie man den Nutzwert möglichst groß machen kann unter der räumlichen oder gewichtsmäßigen Nebenbedingung.

Zahlen-Beispiel 1: $G = 12$, $W = 6$, $n = 4$

Folge der g_i:	4	5	6	9
Folge der zugehörigen w_i:	2	3	3	6

Finden Sie eine geeignete Teilmenge!

Sicher haben Sie rasch eine Lösung gefunden.
Auf der nächsten Folie erscheint eine Lösung:

Zahlen-Beispiel 1: $G = 12$, $W = 6$, $n = 4$

Folge der g_i:	4	5	6	9
Folge der zugehörigen w_i:	2	3	3	6

Teilfolge der g_i:	5	6
Zugehörige Teilfolge der w_i:	3	3

Kontrolle: $5 + 6 = \mathbf{11} \leq G = 12$

$3 + 3 = \mathbf{6} \geq W = 6$

Hinweis 1: Es gibt noch eine weitere Lösung, welche?

Hinweis 2: Setzt man $W = 7$, so gibt es keine Lösung.

Zahlen-Beispiel 2: $G = 40$, $W = 15$, $n = 8$

Folge der g_i:	6	7	8	9	9	11	13	15
Folge der zugehörigen w_i:	2	2	3	4	5	5	6	5

Eine geeignete Teilmenge lautet:

Teilfolge der g_i:	6	7	8	9	9
Zugehörige Teilfolge der w_i:	2	2	3	4	5

Kontrolle:

$6 + 7 + 8 + 9 + 9 = \mathbf{39} \leq G = 40$

$2 + 2 + 3 + 4 + 5 = \mathbf{16} \geq W = 15$

Versuchen Sie, weitere geeignete Teilmenge zu finden!

Gibt es hier weitere Lösungen? Ja, zum Beispiel:

Teilfolge der g_i:	11	13	15
Zugehörige Teilfolge der w_i:	5	6	5

Kontrolle:

$11 + 13 + 15 = \mathbf{39} \leq G = 40$

$5 + 6 + 5 = \mathbf{16} \geq W = 15$

Noch eine Lösung, sogar mit größerer Wichtigkeit:

Teilfolge der g_i:	7	9	11	13
Zugehörige Teilfolge der w_i:	2	5	5	6

Kontrolle:

$7 + 9 + 11 + 13 = \mathbf{40} \leq G = 40$

$2 + 5 + 5 + 6 = \mathbf{18} \geq W = 15$

Es gibt noch mehrere weitere Lösungen, aber keine mit größerer Wichtigkeit als 18. Bitte selbst untersuchen.

Zahlen-Beispiel 3: $G = 300$, $W = 100$, $n = 10$

Folge der g_i:	26 30 42 46 57 60 70 83 88 94
Folge der zugehörigen w_i:	8 6 13 12 15 23 22 30 29 34

Versuchen Sie in 4 Minuten, eine Lösung zu finden!

Eine Lösung hierzu (es gibt weitere):

Teilfolge der g_i:	57	60	88	94
Zugehörige Teilfolge der w_i:	15	23	29	34

Kontrolle:

$57 + 60 + 88 + 94 = 299 \leq G = 300$

$15 + 23 + 29 + 34 = 101 \geq W = 100$

Gibt es weitere Lösungen für dieses Beispiel? Dies lässt sich durch ein systematisches Probierverfahren nachprüfen, das auf folgendem Prinzip beruht:

Man betrachtet einen Gegenstand, z.B. den letzten mit der Nummer k, und ermittelt, wie eine Lösung aussieht, zu der dieser Gegenstand gehört, und wie eine Lösung aussieht, zu der er nicht gehört.

Diese Überlegung führt zu dem folgenden Prinzip der Aufspaltung des Problems in (zwei) Teilprobleme.

Man zerlege das Problem

P

Zu G, W, $g_1, g_2, \ldots, g_k$ und $w_1, w_2, \ldots, w_k$ finde man eine Teilfolge $g_{i_1}, g_{i_2}, \ldots, g_{i_r}$, deren Summe nicht größer als G und deren zugehörige Summe $w_{i_1} + w_{i_2} + \ldots + w_{i_r} \geq W$ ist.

in die beiden Probleme

P1

Zu G, W, $g_1, g_2, \ldots, g_{k-1}$ und $w_1, w_2, \ldots, w_{k-1}$ finde man eine Teilfolge $g_{i_1}, g_{i_2}, \ldots, g_{i_r}$, deren Summe nicht größer als G und deren zugehörige Summe $w_{i_1} + w_{i_2} + \ldots + w_{i_r} \geq W$ ist.

und

P2

Zu $G-g_k$, $W-w_k$, $g_1, g_2, \ldots, g_{k-1}$ und $w_1, w_2, \ldots, w_{k-1}$ finde man eine Teilfolge $g_{i_1}, g_{i_2}, \ldots, g_{i_r}$, deren Summe nicht größer als $G-g_k$ und deren zugehörige Summe $w_{i_1} + w_{i_2} + \ldots + w_{i_r} \geq W-w_k$ ist.

Dieses Prinzip ist unmittelbar klar: Wenn es eine Lösung für P gibt, dann gehört der k-te Gegenstand entweder zur Lösung und dann hat P2 eine Lösung oder er gehört nicht zur Lösung und dann hat P1 eine Lösung. Gibt es andererseits keine Lösung für P, dann können weder P1 noch P2 eine Lösung haben.

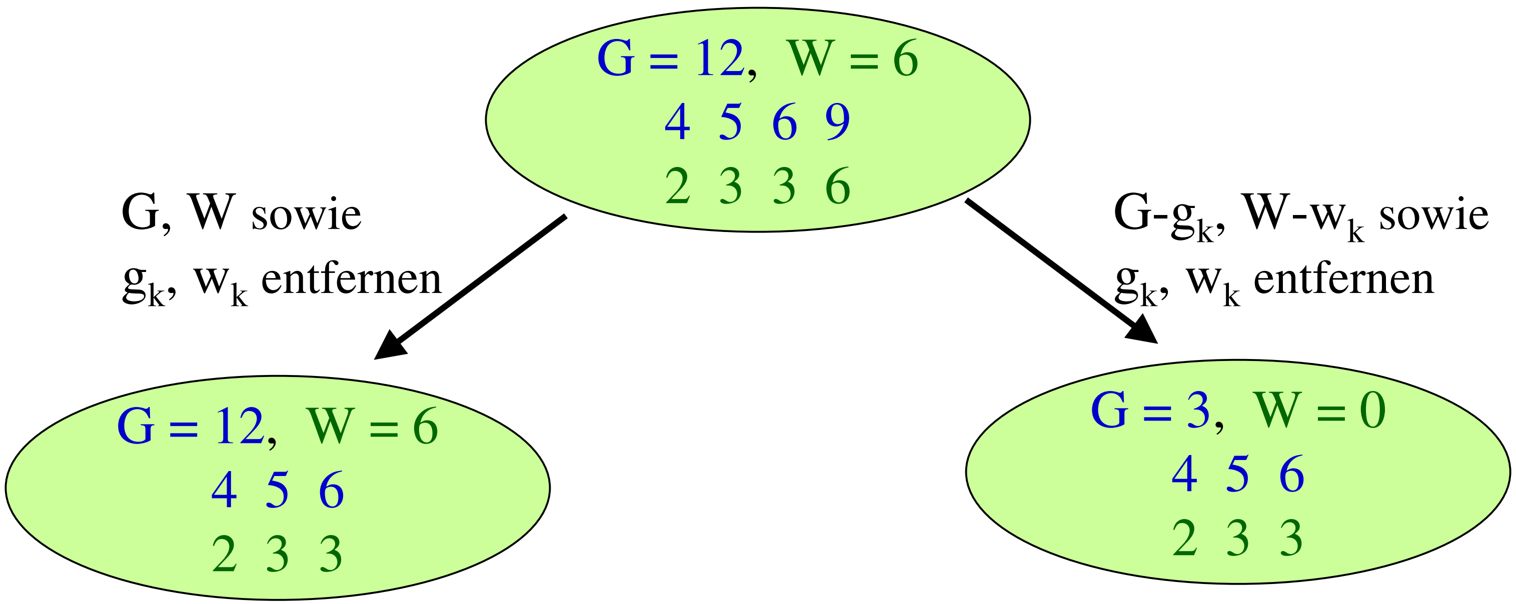
Also kann man die Frage, ob P eine Lösung hat, auf die Frage zurückführen, ob P1 <u>oder</u> P2 eine hat. Genau dieses besagt obiges Prinzip. Man hat damit ein "Problem der Größe k" auf zwei "Probleme der Größe k-1" zurückgeführt.

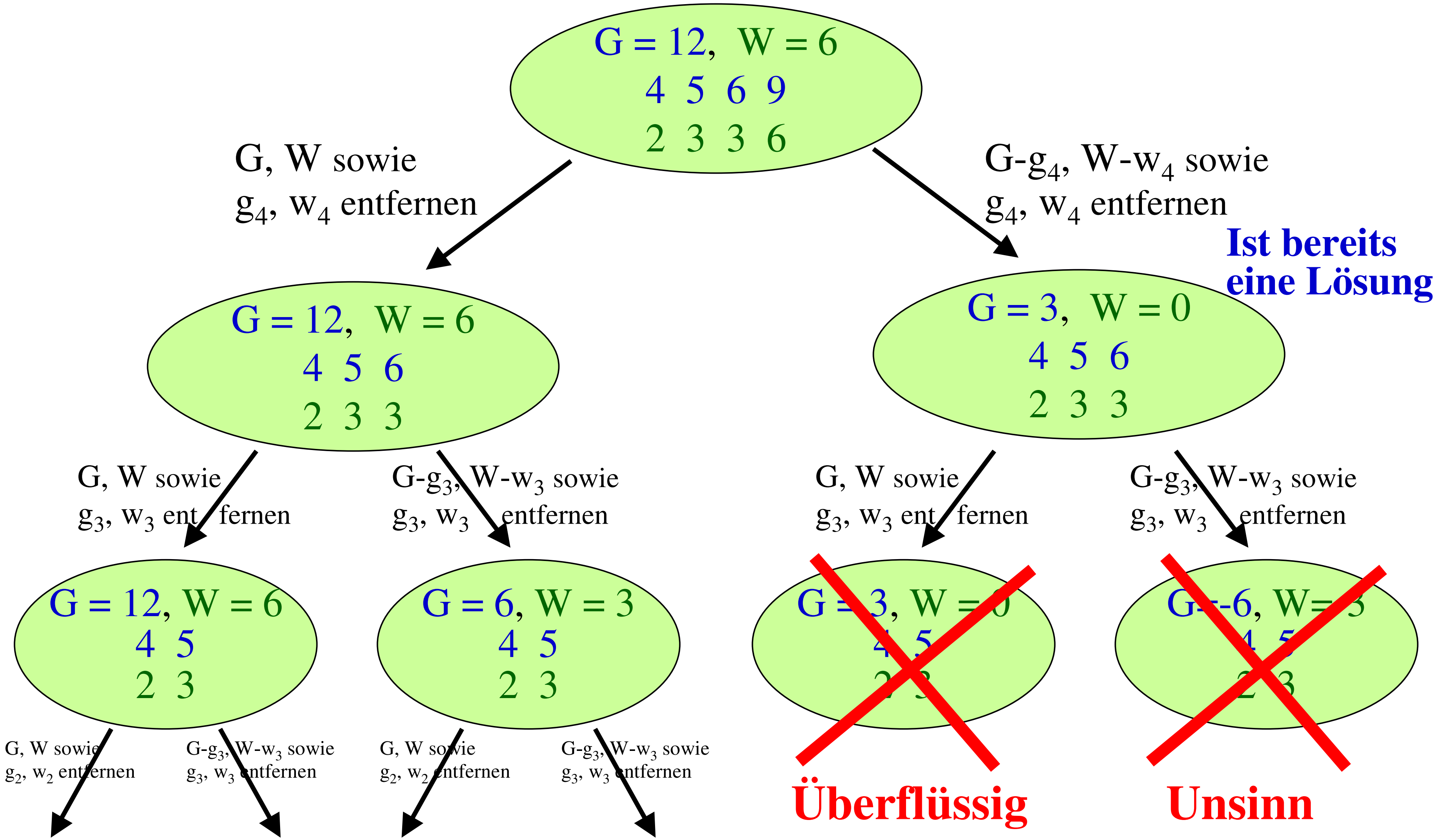
Das zugehörige Verfahren veranschaulicht man durch eine ständige Aufteilung in zwei Unterprobleme; man verfolgt beide Zweige (nacheinander) weiter, bis man unmittelbar entscheiden kann, ob eine Lösung vorliegt oder nicht; dies ist spätestens der Fall, wenn alle Elemente ausgesondert wurden (also für k=0).

Veranschaulichung am Zahlen-Beispiel 1: $G = 12$, $W = 6$, $n = 4$

Folge der g_i: 4 5 6 9

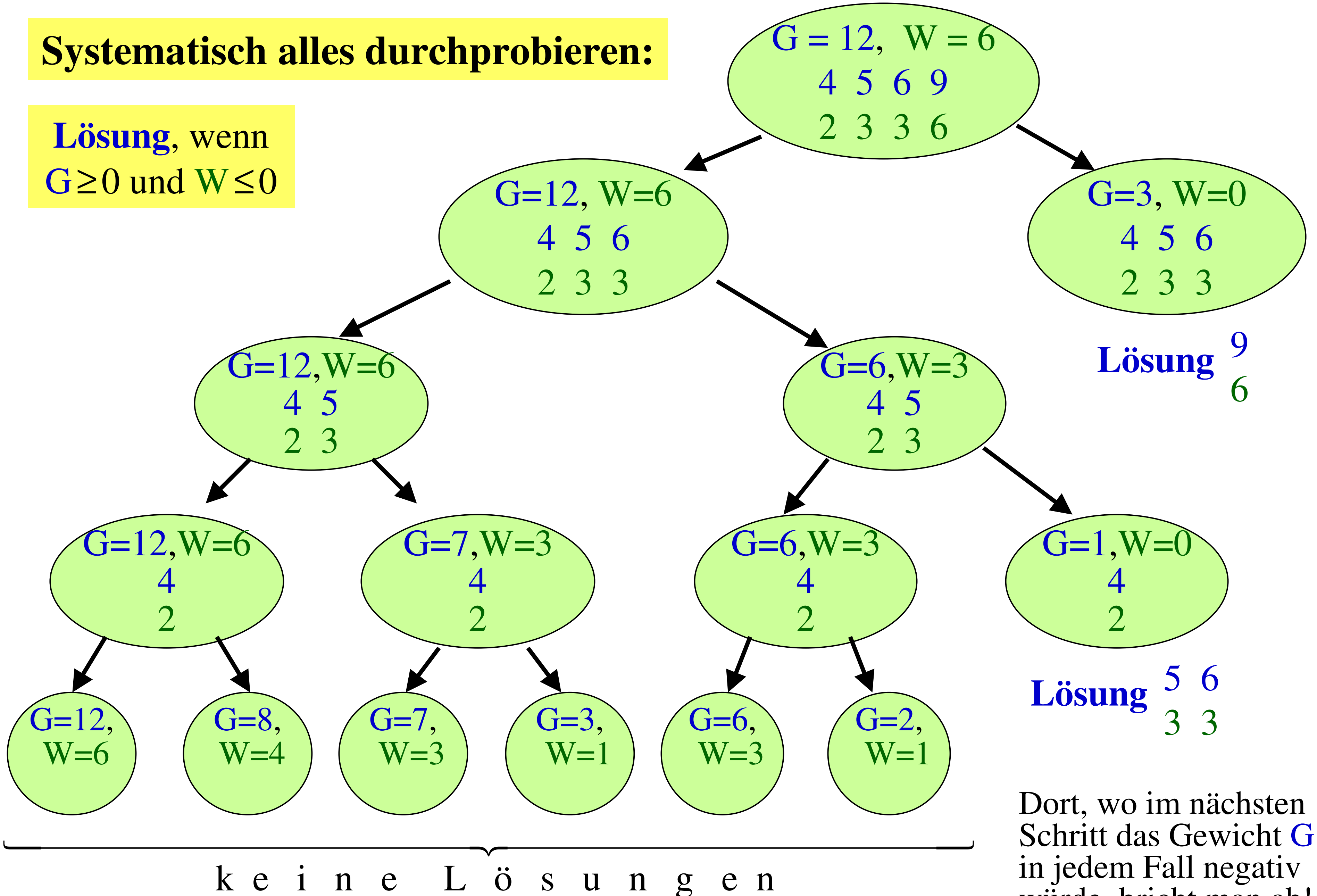
Folge der zugehörigen w_i: 2 3 3 6




G = 12, W = 6
4 5 6 9
2 3 3 6
G, W sowie
g4, w4 entfernen
G-g4, W-w4 sowie
g4, w4 entfernen
Ist bereits eine Lösung
G = 12, W = 6
4 5 6
2 3 3
G = 3, W = 0
4 5 6
2 3 3
G, W sowie
g3, w3 ent fernen
G-g3, W-w3 sowie
g3, w3 entfernen
G, W sowie
g3, w3 ent fernen
G-g3, W-w3 sowie
g3, w3 entfernen
G = 12, W = 6
4 5
2 3
G = 6, W = 3
4 5
2 3
G = 3, W = 0
4 5
2 3
G = -6, W = 3
4 5
2 3
Überflüssig
Unsinn
G, W sowie
g2, w2 entfernen
G-g3, W-w3 sowie
g3, w3 entfernen
G, W sowie
g2, w2 entfernen
G-g3, W-w3 sowie
g3, w3 entfernen

Systematisch alles durchprobieren:

Lösung, wenn $G \geq 0$ und $W \leq 0$



Dort, wo im nächsten Schritt das Gewicht G in jedem Fall negativ würde, bricht man ab!

Ein vollständiges Probierverfahren durchläuft also einen (auf die Spitze gestellten) "Baum", bei dem sich unten in jedem Schritt die Zahl der durchzuprüfenden Fälle verdoppelt. Hin und wieder kann man an einer Stelle abbrechen, weil G negativ wird, aber hierdurch wird der Baum in der Regel nicht wesentlich kleiner. Dieses regelmäßige Verdoppeln führt zu einem "exponentiellen Wachstum" des Baumes und damit zu einer sehr hohen Laufzeit: Wenn man n Gegenstände vorgibt, so muss man mit 2^n Zeiteinheiten rechnen. Auch moderne Computer können dies höchstens bis n=35 in angemessener Zeit durchführen. Für größere Probleme muss man nach anderen Verfahren suchen.

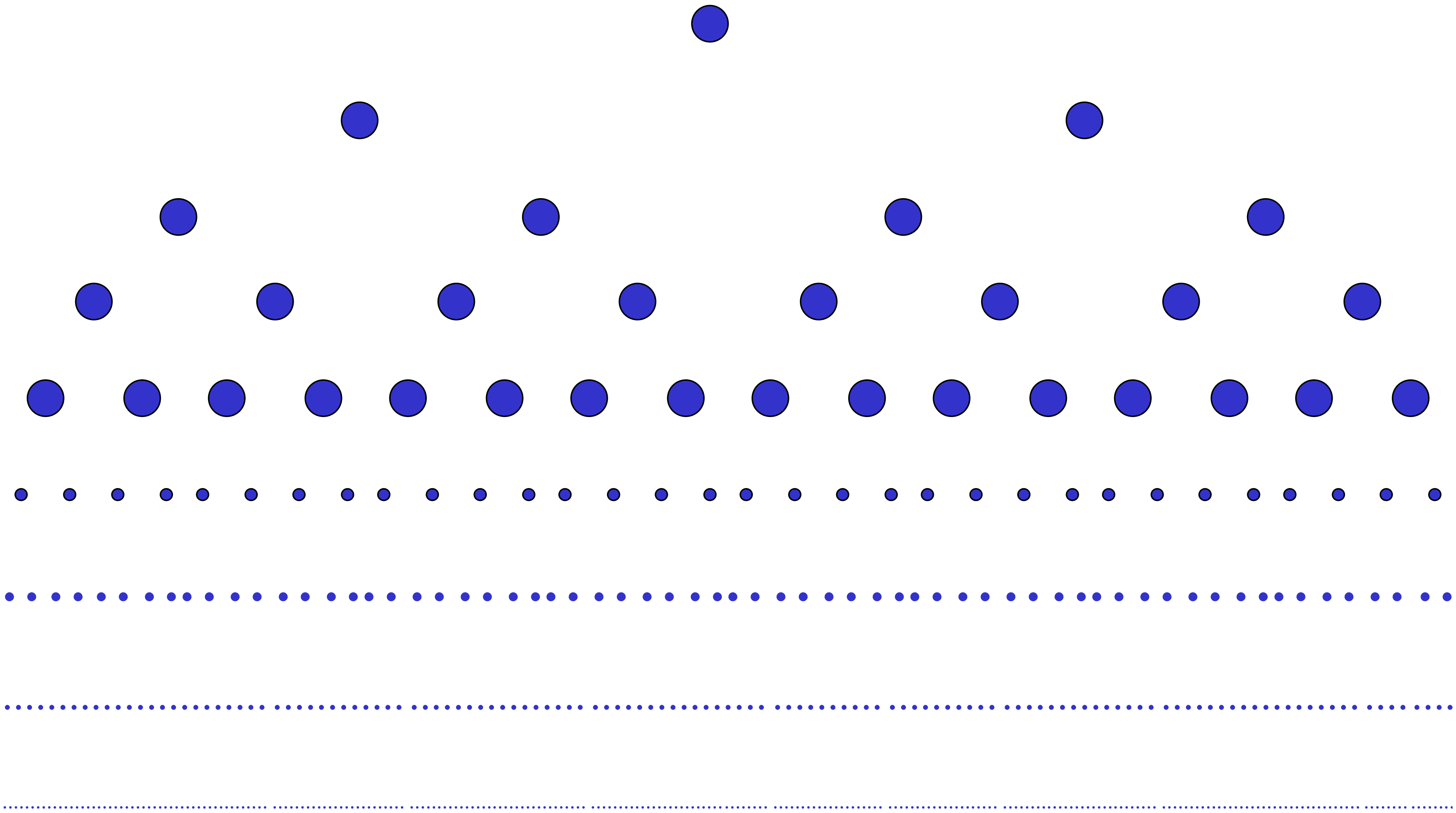
Dieses systematische Durchsuchen aller möglichen Fälle bezeichnet man als **Backtracking** (Rückverfolgen), da man den Baum algorithmisch von rechts nach links durchläuft: Man wählt immer zunächst den rechten Zweig, ohne sich um alles, was links steht, zu kümmern, und sobald man auf den Fall stößt, dass G negativ wird, geht man zurück auf die darüber liegende Ebene und verfolgt dort den linken Zweig weiter.

Um eine Vorstellung von dem Wachstum des Baumes aller möglichen Fälle zu bekommen, ist auf der nächsten Folie die Verdoppelung in jedem Schritt dargestellt. Auf der obersten (der nullten) Ebene startet man mit $2^0 = 1$ Möglichkeiten, dann folgen auf der ersten Ebene $2^1 = 2$ Möglichkeiten, danach auf der zweiten Ebene $2^2 = 4$, dann $2^3 = 8$ usw. Auf der i-ten Ebene befinden sich 2^i Möglichkeiten, so dass bei n Ebenen insgesamt

$$2^0 + 2^1 + 2^2 + ... + 2^i + ... + 2^n = 2^{n+1} -1$$

Stellen in diesem Baum existieren, die auszuwerten sind. Die Entscheidung, ob eine Lösung vorliegt, fällt meist erst auf der untersten (der n-ten) Ebene, auf der sich 2^n Elemente befinden.

Verdoppelung auf jeder Ebene. Hier: neun Ebenen von 2^0 bis 2^8 Elementen:



In unserem Beispiel konnte man bei $G < 0$ abbrechen, weil die nachfolgenden Fälle keine Lösungen mehr erlauben. Will oder muss man in der Praxis ein Backtracking einsetzen, so wird man versuchen, möglichst früh (also auf einer kleinen Ebene) die Fälle zu erkennen, unterhalb derer keine Lösungen mehr möglich sind. Anschaulich: Man muss möglichst frühzeitig die Zweige absägen, die nicht mehr zu einer Lösung führen können. Solche Verfahren bezeichnet man als **Branch-and-Bound**-Verfahren (Verzweige und Begrenze), da hier wie beim Backtracking immer in zwei oder mehr Unterbereiche verzweigt wird, aber bei bestimmten Situationen auch der Unterbaum abgeschnitten und somit der gesamte aufzubauende Baum deutlich begrenzt werden kann.

Branch-and-Bound-Verfahren sind meist stark vom jeweiligen Problem abhängig, so dass wir es bei diesen allgemeinen Bemerkungen belassen wollen.

Definition 5.6: Allgemeines Rucksackproblem

<u>Gegeben</u> seien zwei natürliche Zahlen G und W und eine Menge von n Gegenständen, wobei jeder Gegenstand durch zwei natürliche Zahlen g_i und w_i, sein Gewicht und seine Wichtigkeit, charakterisiert ist.
<u>Frage</u>: Gibt es eine Teilmenge von Gegenständen, so dass die Summe ihrer Gewichte kleiner als G und die Summe ihrer Wichtigkeiten größer als W sind?

Formal:

Gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} \leq G \quad \text{und} \quad \sum_{j=1}^{r} w_{i_j} \geq W \quad ?$$

Spezialfall: Man setze das Gewicht gleich der Wichtigkeit (dies ist der Fall, wenn die Gegenstände beispielsweise Goldklumpen sind). Dann erhält man:

Definition 5.7: Spezielles Rucksackproblem

Gegeben seien eine natürliche Zahl G und eine Folge von n Zahlen $g_1, g_2, ..., g_n$.
Frage: Gibt es eine Teilfolge, deren Summe gleich G ist?
Formal:

Gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G \ ?$$

"**Subset Sum**"

Erneuter Spezialfall: Gerechte Erbschaftaufteilung

Das spezielle Rucksackproblem umfasst zugleich das Erbschaftsproblem: Jemand hinterlässt seinen zwei Erben n Gegenstände im Wert $g_1, g_2, ..., g_n$.
Gibt es eine Aufteilung, so dass beide Erben den gleichen Wert erhalten, d.h., gibt es eine Teilmenge $\{i_1, i_2, ..., i_r\}$ der Menge $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G \quad \text{wobei } G = \left(\sum_{i=1}^{n} g_i\right) / 2 \qquad ?$$

(Falls die Summe der g_i nicht geradzahlig ist, so runde man nach unten oder verbiete solche Gewichte; die volle Schwierigkeit des Problems steckt bereits in den geradzahligen Summen!)

"**Partition**"

In der Literatur werden das spezielle Rucksackproblem auch als "Subset Sum" und das Erbschaftsproblem als "Partition" bezeichnet.

Dieses Erbschaftsproblem sieht harmlos aus; es sind aber bis heute keine Algorithmen bekannt, die im allgemeinen Fall eine Lösung, sofern sie existiert, in weniger als exponentiell vielen Schritten liefert.

Untersuchen Sie ein Beispiel:

Beispiel:

Für das Erbschaftsproblem betrachte folgende 20 natürlichen Zahlen (bereits geordnet aufgeschrieben, ihre Summe ist 1300 und somit G = 650):

$$32, 35, 40, 41, 44, 46, 51, 59, 60, 64,$$
$$72, 75, 76, 78, 80, 85, 86, 89, 92, 95$$

Die Frage lautet also: Man stelle also fest, ob es eine Teilfolge dieser Zahlen gibt, deren Summe 650 ist.

Wer solche Zahlen einige Male untersucht hat, wird spontan behaupten, dass es zu obigem speziellen Beispiel eine Lösung geben wird. Warum?

Erläuterung dieser spontanen Behauptung:
Folgende Überlegung besagt, dass es für unser Beispiel mit recht hoher Wahrscheinlichkeit eine solche Teilfolge gibt: Insgesamt existieren 2^{20}, also ungefähr eine Million Teilfolgen, deren Summen zwischen 0 und 1300 liegen müssen. Es ist sehr unwahrscheinlich, dass die Zahl 650 nicht unter diesen eine Million Zahlen sein sollte.

Damit haben wir eine solche Folge aber noch nicht. Doch bei so hohen Wahrscheinlichkeiten kann man einige Zahlen einfach vorgeben (z.B. 40+60+80+85+95=360) und darauf hoffen, die Ergänzung (also 290) leicht zu entdecken. Durch Probieren findet man diese auch innerhalb kurzer Zeit (z.B. 32, 41, 59, 64, 92 oder 51, 72, 78, 89), so dass z.B. die Teilfolge 32, 40, 41, 59, 60, 64, 80, 85, 92, 95 das Problem löst.

Gefundene Teilfolge (rot): 32,35,40,41,44,46,51,59,60,64,72,75,76,78,80,85,86,89,92,95

Nimmt man dagegen 20 Zahlen, die oberhalb von 2^{30} liegen, dann ist die Chance, dass die Erbschaft sich genau in zwei Teile teilen lässt, i.A. sehr gering; denn dann liegen die eine Million Summen der Teilfolgen ebenfalls oberhalb von 2^{30} (ungefähr eine Milliarde), und dann ist es sehr unwahrscheinlich, dass sich eine vorgegebene Zahl unter diesen Summen befindet.

Wählt man aber die 20 Zahlen so, dass ihre Summe ungefähr 2^{20} beträgt, dann besteht immer noch eine hohe Chance, eine vorgegebene Zahl als Teilsumme darstellen zu können. Bei 2^{21} läge die Wahrscheinlichkeit in der Größenordnung von 0,5.

Fassen wir zusammen: Zu n Zahlen, deren Summe $2 \cdot G$ ist, gibt es 2^n Teilfolgen, deren Summen zwischen 0 und $2 \cdot G$ liegen müssen. Je tiefer G unterhalb von 2^n liegt, um so eher kann man damit rechnen, eine zufällig gewählte Teilfolge, deren Summe kleiner G sei, durch Ausprobieren zu einer Lösung ausbauen zu können. Je stärker G oberhalb von 2^n liegt, um so geringer wird (rein aus der Anzahl der Möglichkeiten betrachtet) die Wahrscheinlichkeit, dass sich die Folge in zwei gleiche Teilsummen aufspalten lässt. Man muss in diesem Fall meist alle Möglichkeiten durchprobieren, vor allem, wenn keine Lösung des Erbschaftsproblems existiert. Das Erbschaftsproblem ist also voraussichtlich dann nur mit dem größtmöglichen Aufwand zu lösen, wenn alle n Zahlen größer als $2^{n+1}/n$ sind, da in diesem Fall sicher $2 \cdot G > 2^{n+1}$, also $G > 2^n$ gilt.

Als Beispiel eignen sich daher besser die folgenden 20 Zahlen, deren Summe 33.480.070 ist, d.h., G = 16.740.035.

1.976.834, 1.864.558, 1.755.621, 1.575.931, 2.169.504,
1.567.429, 2.001.571, 1.682.544, 1.289.337, 1.223.752,
1.884.283, 1.671.449, 1.400.530, 1.547.733, 1.338.626,
1.438.792, 2.010.563, 1.422.589, 1.863.866, 1.794.558

(Wir haben nicht nachgeprüft, ob eine Lösung existiert. Untersuchen Sie diese Zahlen eine Minute lang.)

5.4 Systematisches Durchtesten (backtracking)

Das Backtracking-Verfahren haben wir am Beispiel des Rucksackproblems bereits kennen gelernt. Es führt ein Problem mit Parametern auf sich selbst, aber mit anderen Parametern zurück, wobei die maximale Rekursionstiefe durch die Eingabewerte vorab feststeht.

Erinnerung an Abschnitt 5.3:
Auf Folie 82 steht das Prinzip des Backtrackings speziell für das Rucksackproblem. Wir formulieren es nochmals in sehr allgemeiner kurzer Form:

Man zerlege das Problem

P — P mit Parametern a und k

in die beiden Probleme

P1 — P mit Parametern a_1 und k-1

und

P2 — P mit Parametern a_2 und k-1

Falls eine Terminierungsbedingung erfüllt ist (z.B. k=0, wobei k in irgendeiner Weise die Rekursionstiefe mitzählt), wird die Aufspaltung natürlich nicht mehr durchgeführt, sondern es wird getestet, ob eine Lösung vorliegt usw.
Vor und nach dem rekursiven Aufruf muss man das Problem in der Regel anpassen bzw. diese Anpassung rückgängig machen.

Dies lässt sich unmittelbar als Prozedurschema für das Backtracking (abgekürzt durch BT) formulieren:

```
procedure BT (a, k);
begin
    if  Terminierungsbedingung erfüllt  then  ....
    else   passe das Problem an die erste Rekursion P1 an;
           BT (a1, k-1);
           mache diese Anpassung wieder rückgängig und
           passe das Problem an die zweite Rekursion P2 an;
           BT (a2, k-1);
           mache diese Anpassung wieder rückgängig
    fi
end;
```

Anwenden auf das Rucksackproblem:

Die Variablen G, W, n: natural und g, w: array [1..n] of natural seien global. Sie enthalten die einzugebenden Werte für das Rucksackproblem.
Der Parameter a_1 bzw. a_2 aus dem Schema für das Backtracking ist beim Rucksackproblem das Paar (G, W) bzw. (G-g_k, W-w_k).
Nun wird man hierbei nicht die vorgegebenen Werte für G und W verändern, vielmehr wird man sich zu jedem Zeitpunkt das Gewicht, das noch in den Rucksack maximal hinein gegeben werden kann, merken (Variable *RestG*) sowie die Wichtigkeit, die noch bis zum Erreichen von W fehlt (Variable *RestW*).
Die Parameter sind also (neben der noch verbleibenden Länge k der Folge) RestG, RestW: integer, die anfangs auf den Wert von G bzw. W zu setzen sind. ("integer" statt "natural", da diese Werte beim Subtrahieren negativ werden können.)

Nochmals: Die Variablen G, W, n: <u>natural</u> und g, w: <u>array</u> [1..n] <u>of</u> <u>natural</u> seien global. Die Parameter RestG, RestW, k: <u>integer</u> (anfangs auf den Wert von G bzw. W bzw. n zu setzen) enthalten die aktuellen Werte für das restliche, noch mögliche Gewicht, die bisher im Rucksack noch fehlende Wichtigkeit und (n minus der Tiefe der Rekursion).

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
begin
    if  k=0  then  if  (RestG≥0) and (RestW≤0)  then  "Es gibt eine Lösung"  fi
    else   Rucksack (RestG, RestW, k-1);
           Rucksack (RestG-g[k], RestW-w[k], k-1)
    fi
end;
```

Sobald die globalen Variablen G, W, n, g und w mit den richtigen Werten belegt sind, wird diese Prozedur durch *Rucksack (G, W, n)* aufgerufen.

Diese Prozedur liefert nur eine Ja-Nein-Entscheidung. In der Regel möchte man auch mindestens eine Lösung, d.h., die Teilfolge ermitteln. Hierzu legen wir ein globales Boolesches Feld ja: <u>array</u> [1..n] <u>of</u> <u>Boolean</u> an. Am Ende der Rekursionen, d.h., bei k=0, gibt ja[i] jedes Mal an, ob das i-te Elemente der Folge zur aktuell betrachteten Teilfolge gehört oder nicht.

Das Feld ja wird anfangs mit <u>false</u> initialisiert. Immer, wenn man in die zweite Rekursion verzweigt, in der g[k] und w[k] von RestG bzw. RestW abgezogen werden, d.h., in der das Element k zur Teilfolge hinzu genommen wird, wird ja[k] auf <u>true</u> gesetzt. Kehrt man aus dieser Rekursion zurück, muss ja[k] wieder auf <u>false</u> zurückgesetzt werden.

Zur Erinnerung aus Abschnitt 5.3: Eine Lösung liegt vor, wenn RestG≥0 und RestW≤0 sind; man kann zusätzlich k=0 annehmen (klar).

Globale Variable: G, W, n: natural; g, w: array [1..n] of natural;
ja: array [1..n] of Boolean;

G, W, n und die Felder g und w werden eingelesen; das Feld ja ist anfangs komponentenweise auf false zu setzen.

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
var i: natural;
begin
    if  k=0  then
            if  (RestG≥0) and (RestW≤0)  then  {Lösung gefunden}
                    for i:=1 to n do  if ja[i] then drucke(i) fi  od;
                    {hier: möglicher Abbruch}  fi
    else   Rucksack (RestG, RestW, k-1);
            ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
    fi
end;
```

Will man nur eine Lösung haben, so bricht man die Prozedur an der Stelle *{hier: möglicher Abbruch}* ab. Anderenfalls werden alle Lösungen berechnet.

Dieses Schema des Backtracking wenden wir nun noch auf das spezielle Rucksackproblem an. Zur Erinnerung die Definition:

Gegeben sind eine natürliche Zahl G und eine Folge von n natürlichen Zahlen $g_1, g_2, \ldots, g_n$. Gibt es eine Teilmenge $\{i_1, i_2, \ldots, i_r\}$ der Menge $\{1, 2, \ldots, n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G \quad ?$$

Liest man G nicht ein, sondern setzt man anfangs $G := ½ \cdot \sum_{i=1}^{n} g_i$

so erhält man das Erbschaftsproblem.

Das Programm für das spezielle Rucksackproblem ergibt sich nun unmittelbar aus dem Programm des Rucksackproblems.

Globale Variable: G, n: <u>natural</u>; g: <u>array</u> [1..n] <u>of</u> <u>natural</u>;
ja: <u>array</u> [1..n] <u>of</u> <u>Boolean</u>;

n und das Feld g werden eingelesen, G wird ebenfalls eingelesen bzw. beim Erbschaftsproblem aus g berechnet; das Feld ja ist anfangs komponentenweise auf false zu setzen. Prozedur für das spezielle Rucksackproblem, die mit *SpezRucksack (G, n)* aufgerufen wird:

```
procedure SpezRucksack (RestG: integer, k: integer);
var i: natural;
begin
    if  k=0  then
            if  RestG = 0  then  {Lösung gefunden}
                    for i:=1 to n do  if ja[i] then drucke(i) fi  od;
                    {hier: möglicher Abbruch}  fi
    else   SpezRucksack (RestG, k-1);
            ja[k] := true; SpezRucksack (RestG-g[k], k-1); ja[k] := false
    fi
end;
```

Bei dieser Formulierung des Algorithmus bricht die Rekursion stets erst auf der n-ten Stufe ab. Auch wenn RestG negativ geworden ist, arbeitet das Verfahren weiter, obwohl keine Lösung mehr möglich ist. Wir können die Rekursion sicher abbrechen, wenn RestG kleiner als das Minimum der verbleibenden Gewichte geworden ist. Wenn die Gegenstände nach der Größe ihrer Gewichte geordnet vorliegen, so kann man also die Rekursion auf jeden Fall abbrechen, wenn $RestG < g[1] \leq g[2] \leq \ldots \leq g[k]$ ist, da dann keine Veränderung an RestG mehr statt finden kann.
Dieses Abbruchkriterium für die Rekursion bezog sich auf die Gewichte. Für die Wichtigkeit W gilt etwas Ähnliches: Man kann abbrechen, wenn RestW echt größer ist als die Summe der noch nicht betrachteten Wichtigkeiten, d.h., wenn $RestW > w[1] + w[2] + \ldots + w[k]$ gilt. Um diese Summen nicht jedes Mal auswerten zu müssen, berechnet man sie einmal zu Beginn; man setze also anfangs für $j = 1, 2, \ldots, n$:

$$sumw[j] := w[1] + w[2] + \ldots + w[j]$$

und breche die Rekursion ab, falls $RestW > sumw[k]$ gilt.
Wegen $sumw[1] = w[1]$ und $sumw[j] = sumw[j-1] + w[j]$ lassen sich diese Zahlen sehr schnell berechnen.

Die Abbruchkriterien für die Rekursion lauten also:

$k = 0$

RestG < g[1]   bzw.   RestG < ming (siehe unten)

RestW > sumw[k]

wobei der zweite und dritte Fall stets signalisieren, dass ausgehend von der aktuellen Situation keine Lösung durch weitere Rekursion gefunden werden kann. Für den zweiten Fall ist wichtig, dass die Zahlen sortiert sind. Will man dies nicht voraussetzen, so muss g[1] durch das Minimum "ming" der Menge {g[1], g[2], ..., g[n]} ersetzt werden.

Wir entscheiden uns hier dafür, keine Ordnung vorzugeben und ermitteln daher das Minimum ming zu Anfang mit. Diese Berechnungen lauten also:

```
ming:=g[1]; sumw[1] := w[1];
for j := 2 to n do
        if  ming > g[j]  then  ming := g[j]  fi;
        sumw[j] := sum[j-1] + w[j]
od
```

Damit ergibt sich dann folgendes Programm, das ein spezielles Backtracking, nämlich ein Branch-and-Bound-Verfahren (siehe Abschnitt 5.3) darstellt.

Globale Variable: G, W, n, ming: natural; g, w, sumw: array [1..n] of natural;
ja: array [1..n] of Boolean;

```
procedure Rucksack (RestG: integer, RestW: integer, k: integer);
var i: natural;
begin
    if  k=0  then
            if  (RestG≥0) and (RestW≤0)  then  {Lösung gefunden}
                    for i:=1 to n do  if ja[i] then drucke(i) fi  od;
                    {hier: möglicher Abbruch}  fi
    else if (RestG ≥ ming) and (RestW≤ sumw[k]) then
            Rucksack (RestG, RestW, k-1);
            ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
    fi  fi
end;
begin .... < Einlesen der Werte für n, G, W, g, w >;
for j:=1 to n do ja[j]:=false od; ming:=g[1]; sumw[1] := w[1];
for j := 2 to n do if ming > g[j] then ming := g[j] fi; sumw[j] := sum[j-1] + w[j] od;
Rucksack (G,W,n); ....
end.
```

Nachdem der Algorithmus nun ausformuliert ist, muss man ihn in eine Programmiersprache übertragen. Die Übertragung z.B. nach Ada95 sollte für Sie kein Problem sein.

Hinweise: Bei dieser Übertragung sind Eigenarten der Sprachen zu beachten, z.B. die Angabe von oberen Schranken (statt dynamischer Felder) oder die Tatsache, dass manche Compiler große und kleine Buchstaben nicht unterscheiden (G und g haben also dann den gleichen Bezeichner). Nicht aufgeführt sind die genaue Beschreibung der Dateien, aus denen die Eingaben zu lesen oder in die die jeweiligen Ergebnisse zu schreiben sind.

Weiterhin muss man die Eingabeanweisungen sowie Ergebnis- und Kontrollausdrucke einfügen. In der Praxis ist, wie bereits in Abschnitt 5.3 betont, n=35 wegen der Laufzeit eine obere Grenze für die Anzahl der Gegenstände, es sei denn, die Werte für G und W liegen so günstig, dass durch den Branch-and-Bound-Effekt nur ein kleiner Teil des riesigen Baumes durchsucht werden muss.
Ein Pascal-Programm finden Sie auf der nächsten Folie.

```
program Rucksackproblem; { Programm in PASCAL formuliert }
const  max = 40;
var   n, i, j: int;  Gew, Wert, ming: longint;  g, w, sumw: array[1..max] of longint;
      ja: array [1..max] of Boolean;
procedure Rucksack (RestG, RestW, k: longint);
begin  if  k=0  then
            begin if  ((RestG >= 0) and (RestW <= 0))  then  {Lösung gefunden}
                  begin writeln; for i:=1 to n do  if  ja[i]  then write (i) end; writeln
            end
      else if (RestG ≥ ming) and (RestW≤ sumw[k]) then
__________________begin  Rucksack (RestG, RestW, k-1);
                  ja[k] := true; Rucksack (RestG-g[k], RestW-w[k], k-1); ja[k] := false
                  end;
      if k=n then writeln ('Ende')
end;
begin  read (n);
  if n > max then  writeln ('Eingabe größer als ', max:3, ', Abbruch.')
  else begin read (Gew); read (Wert); writeln; for i:=1 to n do read (g[i], w[i]);
            {Kontrollausgabe:} writeln ('Rucksackproblem mit Gewicht ',Gew:12,' und Wert ',Wert:12,'.');
            writeln ('Es sind ', n:12,' Gegenstände:'); for i:=1 to n do  writeln (g[i]:10, '    ', w[i]:10);
            {Initialisierungen:} for j := 1 to n do ja[j] := false;     ming := g[1]; sumw[1] := w[1];
            for j := 2 to n do begin if ming > g[j] then ming := g[j]; sumw[j] := sum[j-1] + w[j] end;
            Rucksack (Gew, Wert, n)
        end
end.
```

Wir haben das Rucksackproblem behandelt. Die Leser(innen) können durch Weglassen und leichtes Modifizieren leicht Programme für das spezielle Rucksackproblem oder für das Erbschaftsproblem schreiben.
Gibt es weitere Probleme, die man mit Backtracking lösen kann (oder mangels besserer Verfahren lösen muss)?

Am bekanntesten ist das Bin-Packing-Problem (BPP), das wir im nächsten Abschnitt vorstellen werden.

5.5 Binpacking

Ausgehend vom Füllproblem haben wir als eindimensionalen Sonderfall das Rucksackproblem und zwei seiner Vereinfachungen (spezielles Rucksack-problem, Erbschaftsproblem) vorgestellt und als Lösungsmethode das Backtracking herausgearbeitet und als Programm einer Programmiersprache formuliert. Das Backtracking ist eine sehr allgemeine Entwurfsmethode für Lösungsverfahren. Daher gibt es viele andere Probleme geben, die sich hiermit lösen lassen. In diesem Abschnitt wenden wir das Verfahren auf das Binpacking-Problem (BPP) an, eines der älteren Probleme in der Literatur zum Thema Backtracking.

Zunächst gehen wir wieder von einem anschaulichen Problem aus. Als dessen Formalisierung erhalten wir das BPP. Zunächst stellen wir das Binpacking mit anderen Problemen in Beziehung, anschließend lösen wir es mit der Backtracking-Methode.

Problem: Ein Spediteur besitzt **m** Lastwagen des gleichen Typs. Jeder LKW darf mit maximal **G** Gewichteinheiten beladen werden. Eine Kunde möchte **n** Gegenstände, die jeweils das Gewicht $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$ besitzen, zu einem anderen Ort transportieren lassen. Reichen hierfür die m Lastwagen aus? (Hier interessiert nur das Gewicht, nicht die Form der Gegenstände.)

Dieses Problem können wir unmittelbar in eine Definition fassen:

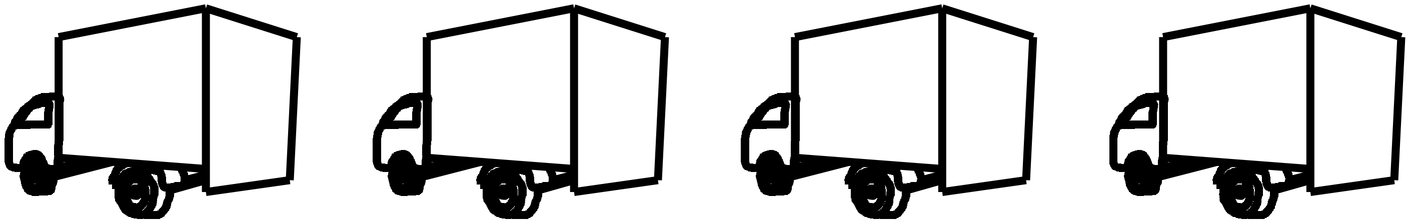
Definition 5.8: Binpacking

Gegeben: natürliche Zahlen **G**, **m**, **n**, $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$.
Frage: Gibt es eine Aufteilung der n Zahlen g_i auf m Teilmengen, so dass die Summe der Zahlen in jeder Teilmenge kleiner oder gleich G ist?

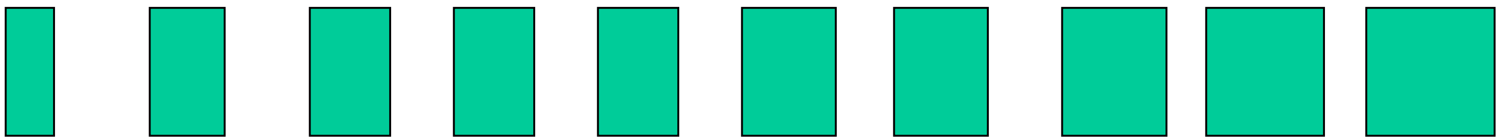
Der Name des Problems kommt vom englischen Wort "bin" = Behälter, Kasten. Diese Behälter sollen optimal "bepackt" werden.

Schauen wir uns das Problem an Beispielen an.

Beispiel 1: $m = 4$, $G = 20$, $n = 10$, die Gewichte der zehn Gegenstände:
$g_1 = 3$, $g_2 = 5$, $g_3 = 6$, $g_4 = 6$, $g_5 = 6$, $g_6 = 7$, $g_7 = 7$, $g_8 = 8$, $g_9 = 9$, $g_{10} = 10$.
Die Gegenstände wiegen insgesamt 67 Einheiten.



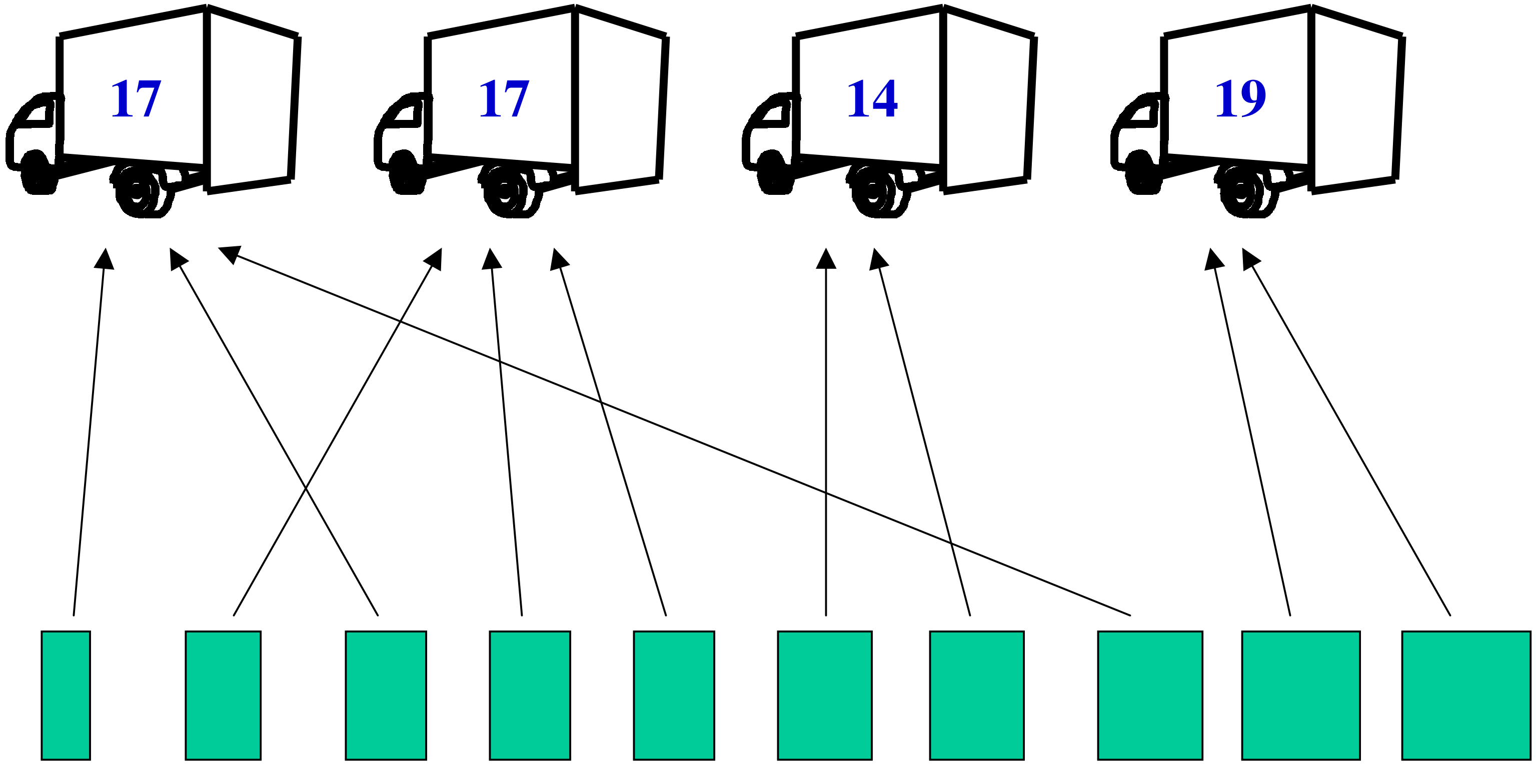
Nun müssen wir die Gegenstände einpacken!



$g_1 = 3$, $g_2 = 5$, $g_3 = 6$, $g_4 = 6$, $g_5 = 6$, $g_6 = 7$, $g_7 = 7$, $g_8 = 8$, $g_9 = 9$, $g_{10} = 10$

Wir ordnen irgendwie von links nach rechts zu:

Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:



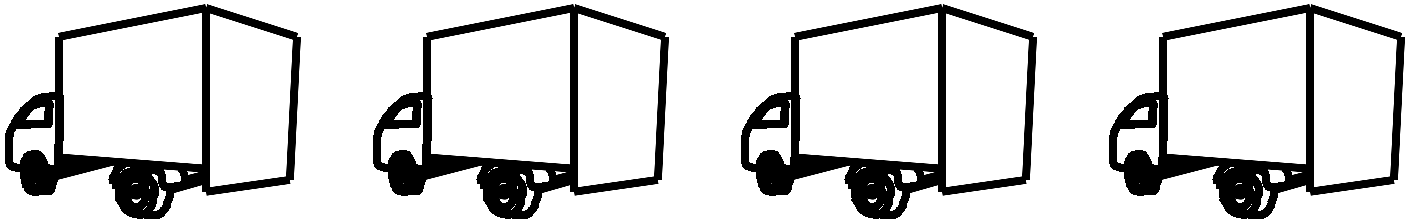
$g_1 = 3$, $g_2 = 5$, $g_3 = 6$, $g_4 = 6$, $g_5 = 6$, $g_6 = 7$, $g_7 = 7$, $g_8 = 8$, $g_9 = 9$, $g_{10} = 10$

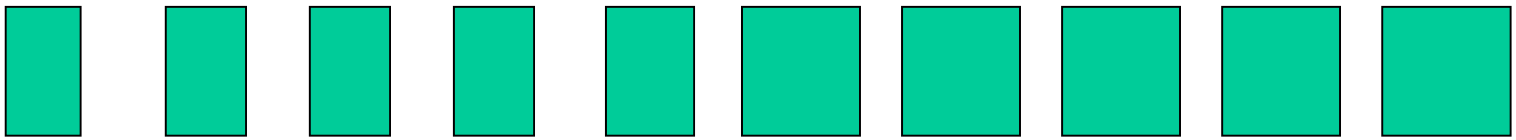
Also haben wir eine Lösung gefunden. (Es gibt noch viele weitere, selbst suchen.)

Beispiel 2: $m = 4$, $G = 20$, $n = 10$, die Gewichte der zehn Gegenstände:

$g_1 = 5$, $g_2 = 6$, $g_3 = 6$, $g_4 = 6$, $g_5 = 7$, $g_6 = 9$, $g_7 = 9$, $g_8 = 9$, $g_9 = 9$, $g_{10} = 10$.

Die Gegenstände wiegen insgesamt 76 Einheiten.



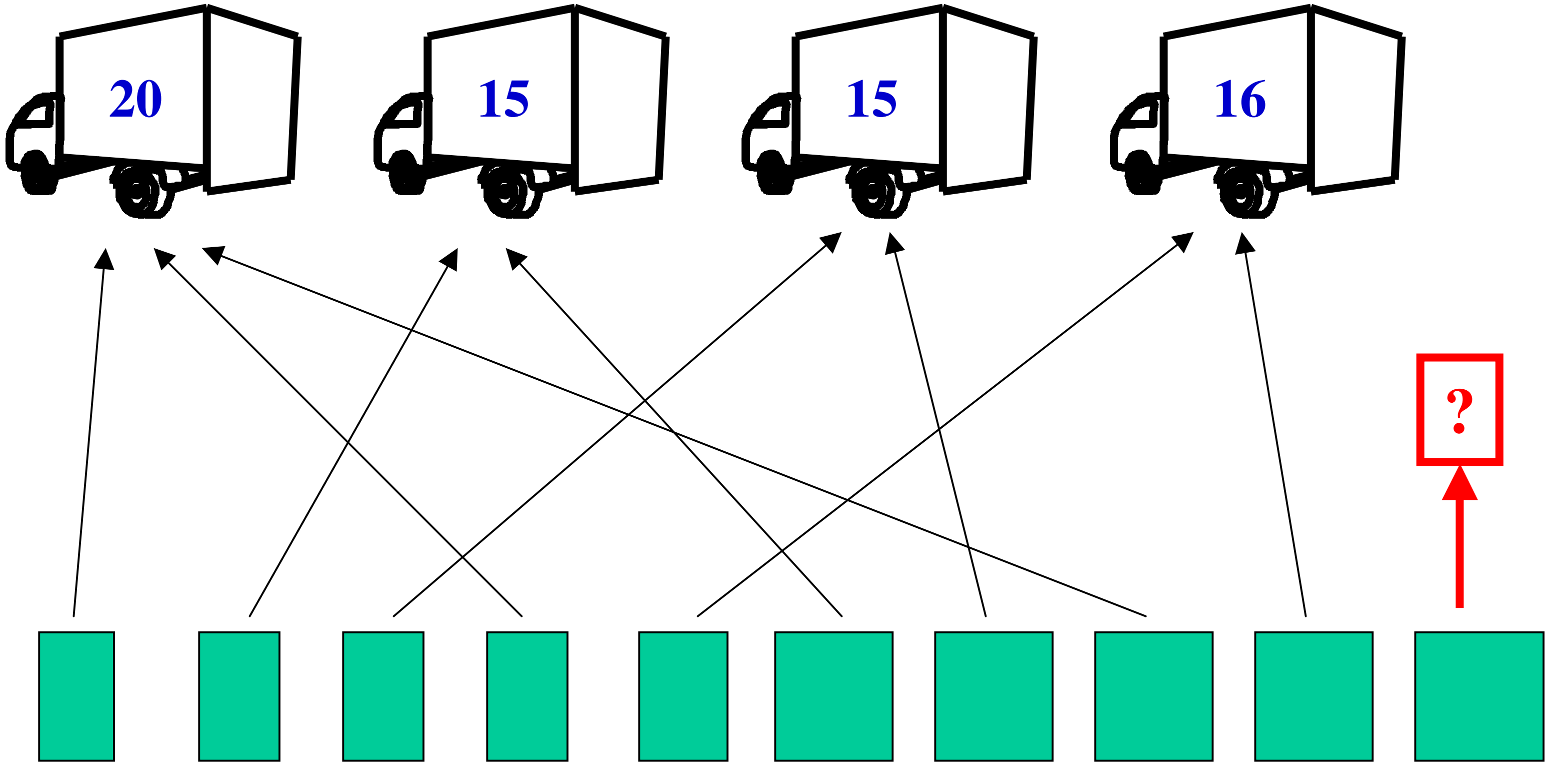


$g_1 = 5$, $g_2 = 6$, $g_3 = 6$, $g_4 = 6$, $g_5 = 7$, $g_6 = 9$, $g_7 = 9$, $g_8 = 9$, $g_9 = 9$, $g_{10} = 10$

Nun versuchen wir, die Gegenstände einzupacken.

Wir ordnen irgendwie von links nach rechts zu:

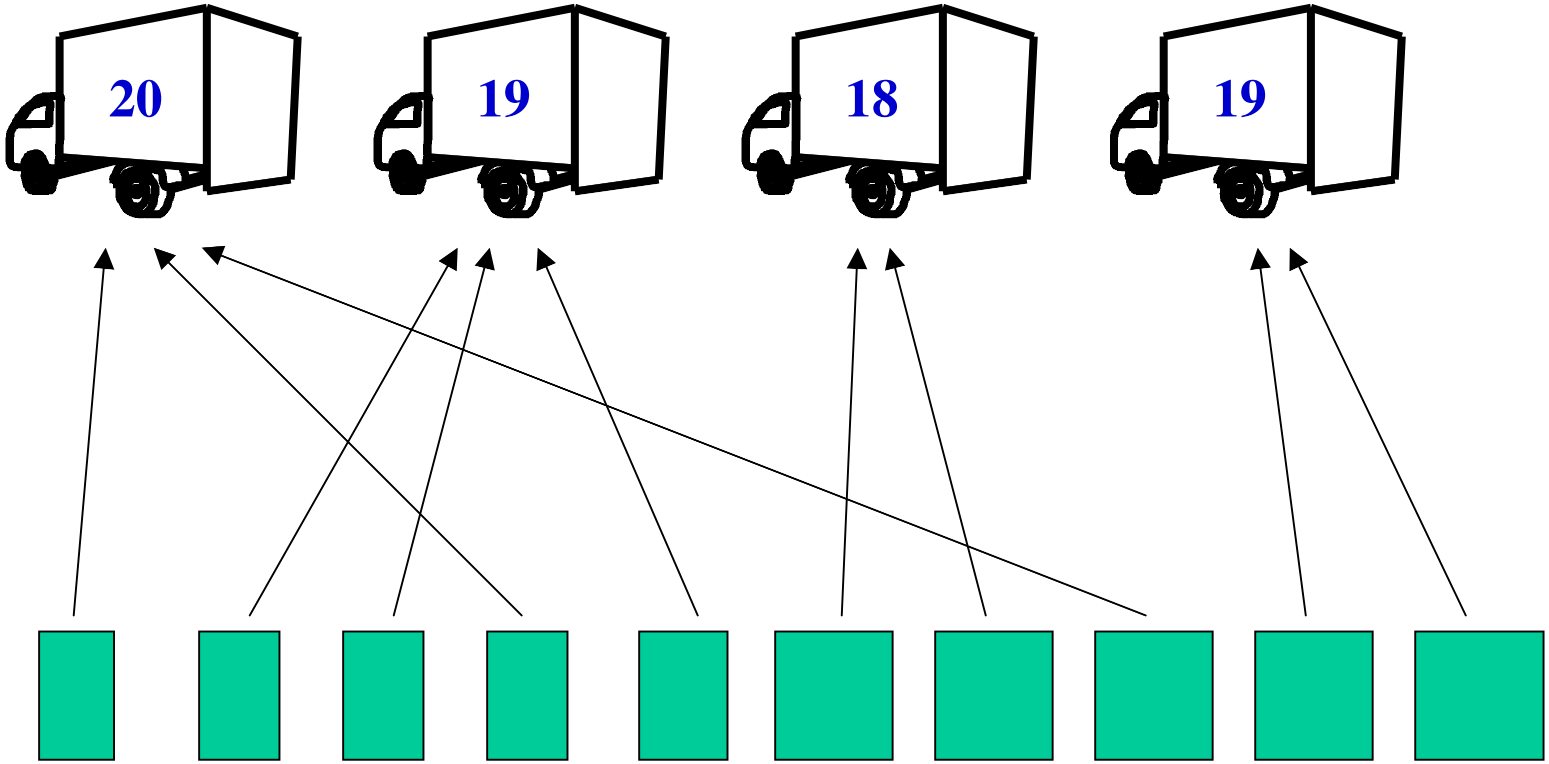
Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:



g_1= 5, g_2= 6, g_3= 6, g_4= 6, g_5= 7, g_6= 9, g_7= 9, g_8= 9, g_9= 9, g_{10}= 10

Die "10" lässt sich hier nicht mehr unterbringen. Aber: Es gibt Lösungen:

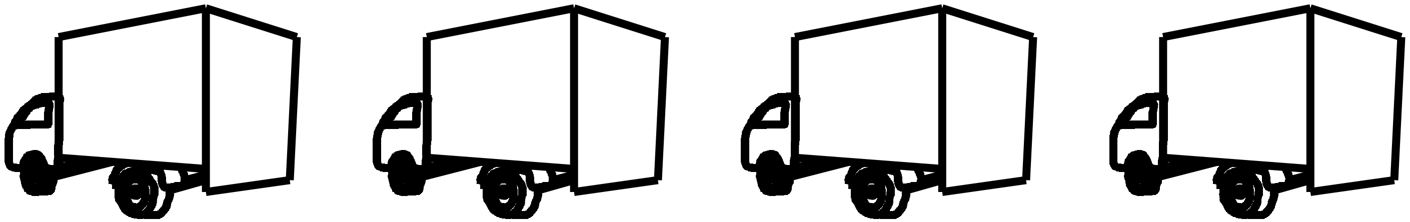
Wir prüfen, ob das Maximalgewicht G=20 irgendwo überschritten wurde:

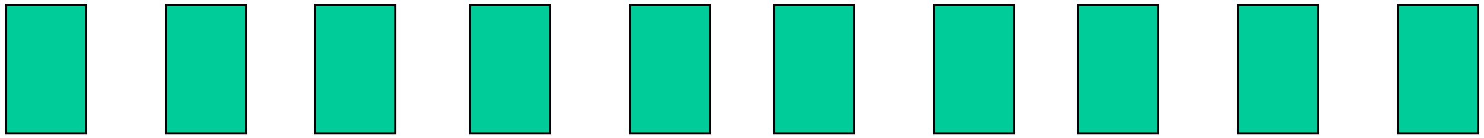


$g_1 = 5$, $g_2 = 6$, $g_3 = 6$, $g_4 = 6$, $g_5 = 7$, $g_6 = 9$, $g_7 = 9$, $g_8 = 9$, $g_9 = 9$, $g_{10} = 10$

Somit haben wir eine Lösung für Beispiel 2 gefunden.

<u>Beispiel 3</u>: $m = 4$, $G = 20$, $n = 10$, die Gewichte der zehn Gegenstände:
$g_1 = 7$, $g_2 = 7$, $g_3 = 7$, $g_4 = 7$, $g_5 = 7$, $g_6 = 7$, $g_7 = 7$, $g_8 = 7$, $g_9 = 7$, $g_{10} = 7$.
Die Gegenstände wiegen insgesamt 70 Einheiten.





$g_1 = 7$, $g_2 = 7$, $g_3 = 7$, $g_4 = 7$, $g_5 = 7$, $g_6 = 7$, $g_7 = 7$, $g_8 = 7$, $g_9 = 7$, $g_{10} = 7$

Beim Einpacken werden wir offensichtlich scheitern; denn in jeden LKW passen wegen $G = 20$ höchstens zwei Gegenstände.

Beispiel 3 hat also keine Lösung.

Zur Einordnung des Problems:

Das BPP verallgemeinert das Rucksackproblem von einem auf viele Rucksäcke.
Wir zeigen nun, dass das spezielle Rucksackproblem (vgl. Definition 5.7) und das Erbschaftsproblem Sonderfälle des BPP für m=2 Behälter sind.

Hilfssatz 5.1:

Es seien eine natürliche Zahl G und eine Folge von n Zahlen $g_1, g_2, ..., g_n$ gegeben. Bilde die Summe aller Gewichte $SG = g_1 + g_2 + ... + g_n$. Ohne Beschränkung der Allgemeinheit sei $2 \cdot G \leq SG$. Dann gilt:

Das spezielle Rucksackproblem besitzt für die Werte $G, n, g_1, g_2, ..., g_n$ (mit $2 \cdot G \leq SG$) genau dann eine Lösung, wenn das Binpackingproblem für $SG\text{-}G, 2, n+1, g_1, g_2, ..., g_n, g_{n+1}$ mit $g_{n+1} = SG - 2 \cdot G$ eine Lösung hat.

Beweis:

Wir beweisen zunächst den Hilfssatz und erläutern dann, warum man *ohne Beschränkung der Allgemeinheit* $2 \cdot G \leq SG$ annehmen kann.

Beweisrichtung "⇒":

Das spezielle Rucksackproblem für G, n, $g_1, g_2, ..., g_n$ möge eine Lösung haben, und zwar sei die Gewichtssumme der Teilmenge $L=\{g_{i_1}, g_{i_2}, ..., g_{i_r}\}$ genau G. Wir füllen nun die Gegenstände dieser Teilmenge zusammen mit dem (n+1)-ten Gegenstand in den ersten der beiden Behälter. Ihr Gewicht ist:

$g_{i_1} + g_{i_2} + ... + g_{i_r} + g_{n+1} = G + (SG-2 \cdot G) = SG-G,$

der erste Behälter wird hierdurch also bis zu seinem Maximalgewicht gefüllt.

Alle übrigen Gegenstände füllen wir in den zweiten Behälter. Das Gewicht aller dieser Gegenstände beträgt SG-G. Hierdurch wird also auch der zweite Behälter bis zu seinem Maximalgewicht (SG-G) aufgefüllt.

Diese Aufteilung auf die beiden Behälter ist folglich eine Lösung des Binpackingproblems für SG-G, 2, n+1, $g_1, g_2, ..., g_n, g_{n+1}$ mit $g_{n+1} = SG- 2 \cdot G$; was zu beweisen war.

Beweisrichtung "$\Leftarrow$":

Wenn eine Lösung des BPP für SG-G, 2, n+1, g_1, g_2, ..., g_n, g_{n+1} mit g_{n+1} = SG-2·G vorliegt, dann lassen sich die n+1 Gegenstände auf zwei Behälter mit jeweils maximalem Gewicht SG-G aufteilen. Dieses Maximalgewicht wurde geschickt gewählt, denn es gilt:

$$2\cdot(SG\text{-}G) = SG - 2\cdot G + SG = g_{n+1} + g_1 + g_2 + ... + g_n.$$

Also werden durch die n+1 Gegenstände beide Behälter bis zu ihrem Maximalgewicht gefüllt.

In einem der beiden Behälter muss der (n+1)-te Gegenstand mit dem Gewicht g_{n+1} = SG- 2·G liegen. Da er ganz gefüllt ist, muss das restliche Gewicht in diesem Behälter

$$(SG\text{-}G) - g_{n+1} = (SG\text{-}G) - (SG\text{-}2\cdot G) = G$$

sein. Folglich muss es eine Teilmenge der n Gegenstände geben, deren Gewichtssumme gleich G ist, d.h., das spezielle Rucksackproblem für G, n, g_1, g_2, ..., g_n hat eine Lösung; was zu beweisen war.

Damit ist Hilfssatz 5.1 bewiesen.

Nachtrag: Wir hatten behauptet, dass man ohne Beschränkung der Allgemeinheit $2 \cdot G \leq SG$ annehmen dürfe. Begründung hierfür:

Es seien eine natürliche Zahl G und eine Folge von n Zahlen $g_1, g_2, ..., g_n$ gegeben. Bilde die Summe aller Gewichte $SG = g_1 + g_2 + ... + g_n$. Wenn dieses spezielle Rucksackproblem eine Lösung besitzt, dann gilt: Es gibt eine Teilmenge $L = \{i_1, i_2, ..., i_r\}$ der Zahlen $\{1, 2, ..., n\}$ mit

$$\sum_{j=1}^{r} g_{i_j} = G.$$

Dann gilt für die anderen Gegenstände: $\sum_{i \notin L} g_i = SG\text{-}G.$

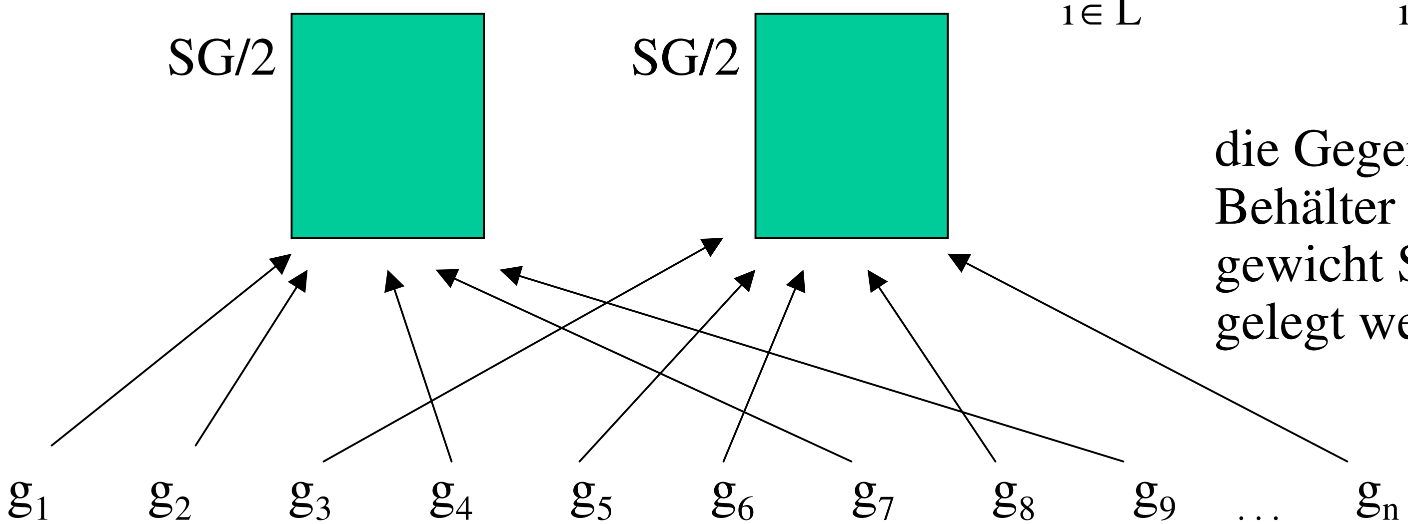
Also bildet eine Lösung bzgl. des Maximalgewichts G zugleich eine Lösung bzgl. des Maximalgewichts SG-G, indem man die komplementäre Teilmenge $\{1, 2, ..., n\}\text{-} \{i_1, i_2, ..., i_r\}$ verwendet. Wenn daher $2 \cdot G > SG$ ist, so löse man stattdessen das Problem mit dem Maximalgewicht SG-G, denn hierfür gilt: $2 \cdot (SG\text{-}G) \leq SG+SG\text{-}2 \cdot G < SG$, d.h., dann ist die geforderte Bedingung erfüllt.

Bemerkung:

Beim Erbschaftsproblem ist sofort klar, dass es ein Binpackingproblem mit 2 Behältern und der Behältergröße SG/2 ist. Denn hier wird festgestellt, ob sich eine Folge von n Zahlen so in zwei Teilfolgen zerlegen lässt, dass deren Summen gleich sind. Formal: Sei für eine Folge von n Zahlen $g_1, g_2, \ldots, g_n$ die Summe aller Gewichte $SG = g_1 + g_2 + \ldots + g_n$, so hat das Erbschaftsproblem genau dann eine Lösung, wenn das Binpackingproblem für $SG/2, 2, n, g_1, g_2, \ldots, g_n$ eine Lösung hat. Denn: Es gibt eine Teilmenge $L = \{i_1, i_2, \ldots, i_r\}$ der Zahlen $\{1, 2, \ldots, n\}$ mit

$$\sum_{i \in L} g_i = SG/2 = \sum_{i \notin L} g_i$$

wenn



die Gegenstände in 2 Behälter mit Maximalgewicht SG/(2) gelegt werden können.

Was besagt der Hilfssatz 5.1 ?

Er vergleicht die beiden Probleme "spezielles Rucksackproblem" und "Binpacking mit 2 Behältern", wobei er nachweist, dass das Binpacking mit 2 Behältern mindestens so schwierig wie das spezielle Rucksackproblem ist.

Anders ausgedrückt: Wenn es ein Verfahren gibt, das das Binpacking mit 2 Behältern schnell löst, so lässt sich das spezielle Rucksackproblem ebenfalls mit diesem Zeitaufwand lösen.

Der Beweis zeigt zugleich, wie sich <u>jedes</u> spezielle Rucksackproblem in das Binpackingproblem einbetten lässt. (Sog. "Reduktion" des einen Problems auf das andere; vgl. später unter Komplexitätstheorie.)

Wir wenden uns nun der Programmierung des Backtrackingverfahrens für das Binpacking zu.

Hierzu müssen wir eine mögliche präzise Definition haben. Die Definition des Binpackings lautete: Finde eine Aufteilung von n Zahlen auf höchstens m Teilmengen, deren Summe jeweils kleiner oder gleich G ist. Wir formulieren dies nun nochmals mathematisch:

Definition 5.8 a: Binpacking

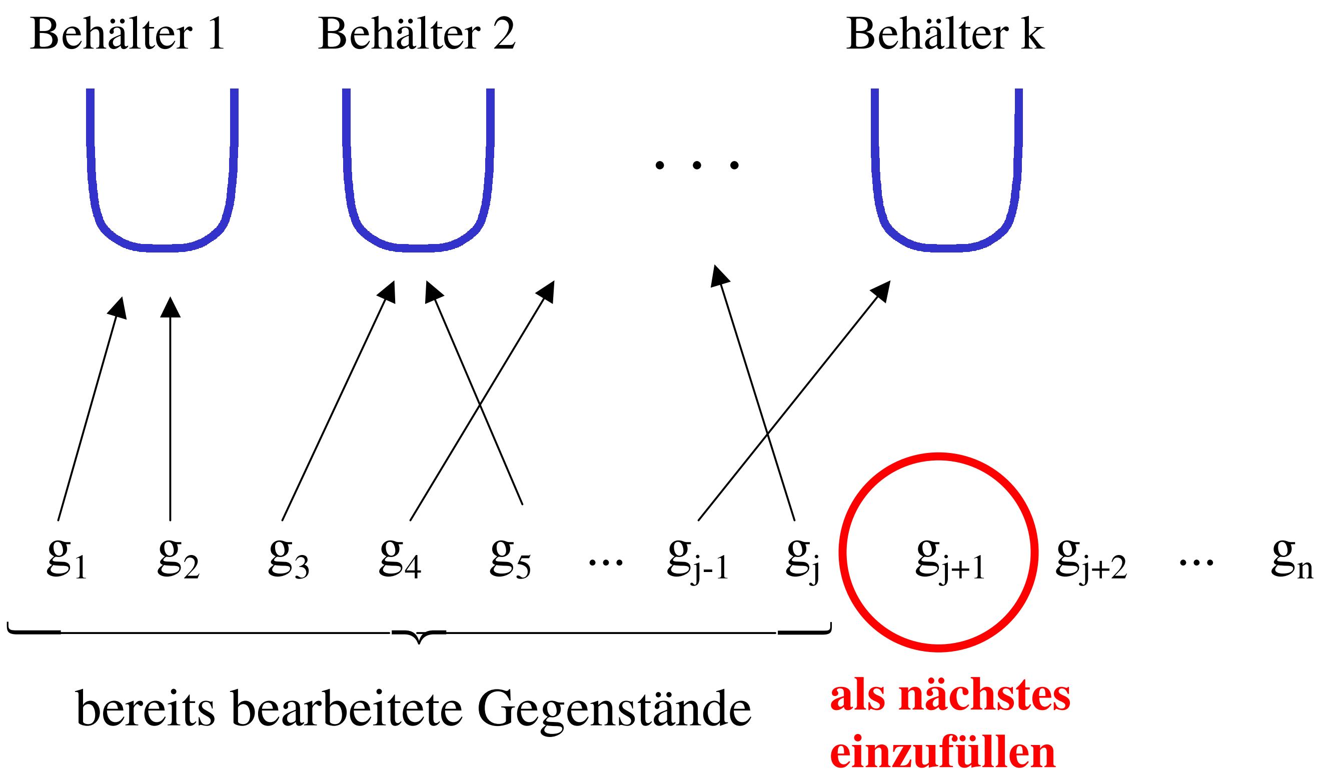
Gegeben: natürliche Zahlen $\mathbf{G}$, $\mathbf{m}$, $\mathbf{n}$, $\mathbf{g_1}$, $\mathbf{g_2}$, ..., $\mathbf{g_n}$.
Gesucht: eine Abbildung $f: \{1, 2, ..., n\} \rightarrow \{1, 2, ..., m\}$,
so dass für jedes i ($1 \leq i \leq m$) gilt:

$$\sum_{f(j)=i} g_j \leq G.$$

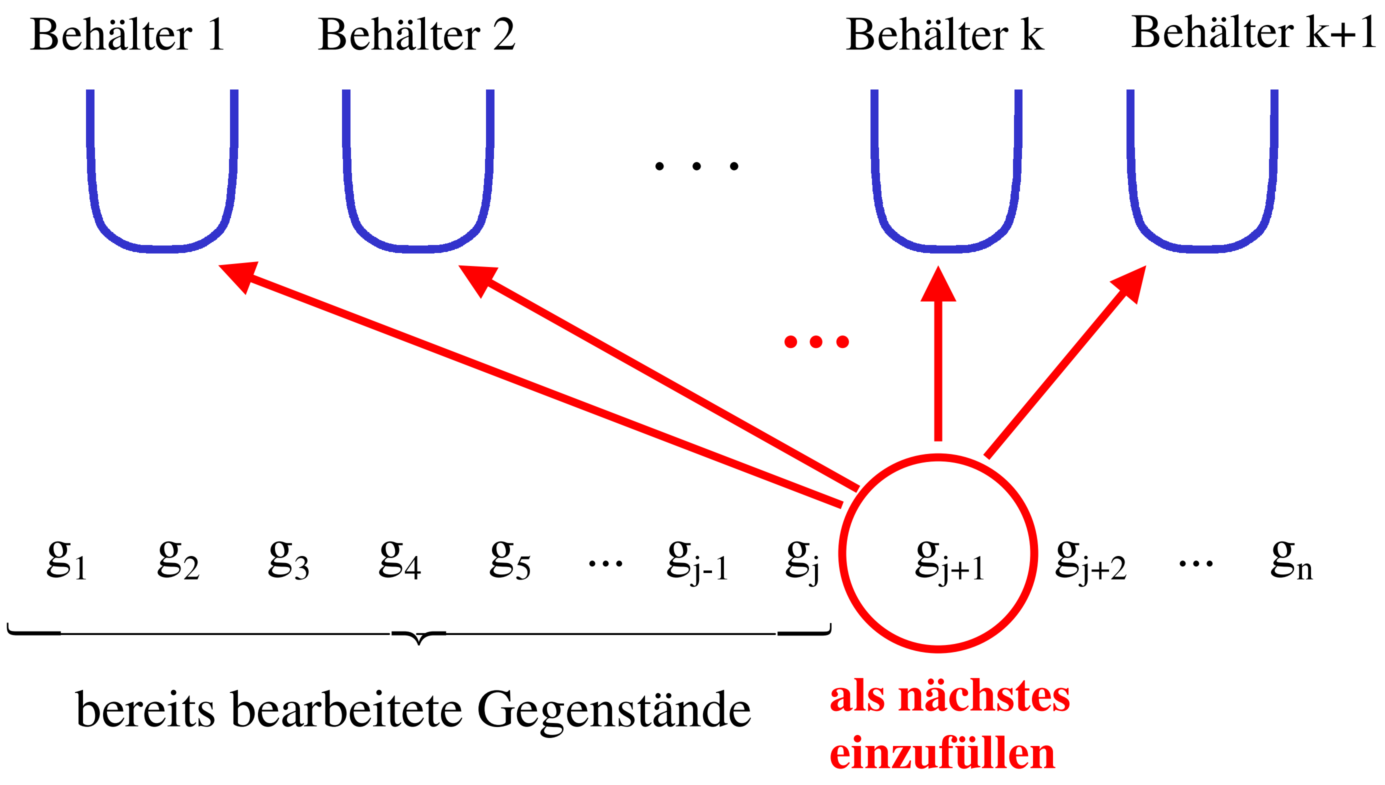
f(j) = i bedeutet also: Der j-te Gegenstand wird in den i-ten Behälter gelegt.

Um eine Lösung zu finden, probieren wir alle Abbildungen durch, sofern sie eine Lösung bilden könnten. Hierfür verwenden wir erneut das Backtracking.

Backtracking geht stets von einer vorhandenen Situation aus und reduziert das Problem auf einfachere Probleme. Eine typische Situation beim Füllen der Behälter ist: Man hat die ersten j Gegenstände bereits konfliktfrei in k Behälter gefüllt und muss nun den (j+1)-ten Gegenstand einfüllen.



Wir können den (j+1)-ten Gegenstand nun einem der bereits vorhandenen Behälter 1, 2, ..., k zuordnen oder einen neuen Behälter (k+1) verwenden, sofern $k+1 \leq m$, d.h., $k < m$ ist.



Alle diese k+1 Fälle probieren wir im Backtracking durch. Die Prozedur folgt genau der obigen Rekursion.

Schema der Prozedur (es fehlen noch einige Details):

```
procedure BTBPP (j, k: natural);
var i: natural;
begin
        if  j = n  then  ..... {Lösung gefunden} .....
        else  for i:=1 to k do
                      if  Gegenstand (j+1) passt noch in Behälter i  then
                             Aufruf von BTBPP (j+1, k)  fi  od;
               if k < m then Aufruf von BTBPP (j+1, k+1) fi
        fi
end;
```

Um zu überprüfen, ob ein Gegenstand j(j+1) noch in den Behälter i passt, verwenden wir ein globales Feld GBeh: array[1..m] of natural, in welchem wir die Summe der Gewichte aller Gegenstände, die aktuell im jeweiligen Behälter liegen, notieren. Ein Gegenstand (j+1) darf in den Behälter i nur gelegt werden, wenn dadurch das Maximalgewicht nicht überschritten wird, also nur, wenn $GBeh[i]+g[j+1] \leq G$ ist.

So erhalten wir die Verfeinerung des Prozedurschemas (man beachte, dass vor und nach jedem rekursiven Aufruf das Gewicht des jeweiligen Behälters angepasst werden muss):

```
procedure BTBPP (j, k: natural);
var i: natural;
begin
        if  j = n  then  ..... {Lösung gefunden} .....
        else  for i:=1 to k do
                        if  GBeh[i] + g[j+1] ≤ G  then
                               GBeh[i]:=GBeh[i]+g[j+1]; BTBPP (j+1, k);
                               GBeh[i]:=GBeh[i]-g[j+1]  fi  od;
               if k < m then GBeh[k+1]:=g[j+1]; BTBPP (j+1, k+1);
                               GBeh[k+1]:=0  fi
        fi
end;
```

Diese Prozedur stellt bisher nur fest, ob eine Lösung existiert, aber sie gibt keine mögliche Zuordnung zu den Behältern aus. Hierfür müssen wir uns noch zu jedem Gegenstand merken, in welchen Behälter er gelegt wurde. Dies geschieht in einem globalen Feld f: <u>array</u> [1..n] <u>of</u> natural. Wir nennen dieses Feld f, weil es genau die Abbildung f aus Definition 5.8.a beschreibt.

Wir brauchen nicht alle möglichen Abbildungen f durchzuprobieren; denn wir können annehmen, dass der Gegenstand mit der Nummer 1 stets im Behälter 1 liegt, der Gegenstand mit der Nummer 2 stets in einem der Behälter 1 oder 2 usw. Durch Umnummerieren der Behälter lässt sich also stets erreichen, dass $f[j] \leq j$ für alle $j=1,2,...,n$ gilt. Wir werden daher $f[1]:=1$ setzen. In der Prozedur stellen wir automatisch sicher, dass $f[j] \leq j$ für alle weiteren j gilt, indem für den Gegenstand j+1 neben den bereits betrachteten Behältern nur der Behälter k+1 (und nicht k+2, k+3 usw.) ausprobiert wird. Weil m die höchste Nummer eines Behälters ist, brauchen wir also nur Abbildungen f zu betrachten mit:

$f(j) \leq \mathrm{Min}(j,m)$, wobei $\mathrm{Min}(j,m)$ das Minimum der beiden Zahlen j und m ist.

So erhalten wir als Backtrackingprozedur für das Binpackingproblem:

```
procedure BTBPP (j, k: natural);
var i: natural;
begin
        if  j = n  then  < Lösung gefunden: drucke das Feld f aus >
        else  for i:=1 to k do
                     if  GBeh[i] + g[j+1] ≤ G  then
                            f[j+1]:=i; GBeh[i]:=GBeh[i]+g[j+1]; BTBPP (j+1, k);
                            GBeh[i]:=GBeh[i]-g[j+1]; f[j+1]:=0  fi  od;
              if k < m then f[j+1]:=k+1; GBeh[k+1]:=g[j+1];
                            BTBPP (j+1, k+1); GBeh[k+1]:=0; f[j+1]:=0  fi
        fi
end;
```

Globale Variablen sind:

G, m, n: natural;
g, f: array[1..n] of natural;
GBeh: array[1..m] of natural;

Der Aufruf der Prozedur BTBPP lautet, sofern bereits alle Daten in die globalen Variablen G, m, n und g eingelesen wurden:

```
if (n<1) or (m<1) then ...{Abbruch, da keine Lösung zu suchen ist}... fi;
for i:=1 to n do if g[i]>G then ...{Abbruch, da keine Lösung möglich}... fi od;
for i:=1 to n do f[i]:=0; GBeh[i]:=0 od;
GBeh[1]:=g[1]; f[1]:=1; BTBPP(1,1);
```

Manche der obigen Anweisungen erscheinen überflüssig (z.B. f[j+1]:=0 oder for i:=1 to n do f[i]:=0; GBeh[i]:=0 od;). Wir haben sie dennoch aufgeführt, da sie eventuell sehr hilfreich werden können, wenn Fehler auftreten oder wenn die Prozedur noch von anderen Programmen aufgerufen wird.

Eigenschaften der Prozedur BTBPP

Wir haben bereits gesehen, dass nur Lösungen mit

$$f(j) \leq \mathrm{Min}(j,m) \quad \text{für } j=1,\ldots,n$$

untersucht werden.

Weiterhin werden wir erwarten, dass die Prozedur die Behälter stets von links nach rechts wirklich auffüllt, dass sich unter den ersten k Behältern (falls der Behälter k nicht leer ist) also niemals ein leerer Behälter befindet; denn sonst würden wir Situationen mehrmals testen, die bis auf die Nummerierung der Behälter gleich sind. Ist durch unsere Prozedur sicher gestellt, dass unter den ersten k Behältern keine leeren auftreten?

Dies ist tatsächlich der Fall, wie folgende Überlegung zeigt.

Zu Beginn wird in Behälter 1 durch GBeh[1]:=g[1] der erste Gegenstand gelegt. Er wird hieraus nicht wieder entfernt. Wir nehmen nun an, dass die ersten j Gegenstände in den Behältern von 1 bis k liegen und dass keiner dieser k Behälter leer ist (Induktionsannahme). Diese Annahme ist für j=1 richtig, wobei hier automatisch k=1 ist. Die Prozedur versucht nun, den nächsten Gegenstand j+1 in einen bereits existierenden Behälter zu legen (for i:=1 to k do ... od). Dabei kann kein vorhandener Behälter geleert werden; in diesem Fall bleibt daher die Induktionsannahme richtig. Anschließend wird der Gegenstand j+1 in den neuen Behälter (k+1) gelegt, sofern k < m ist. Dieser (k+1)-te Behälter ist also nicht leer, so dass die Induktionsannahme auch für den Fall j+1 richtig bleibt. (Nach der Rückkehr aus diesem letzten Rekursionsfall wird der Behälter k+1 geleert, aber k+1 wieder durch k ersetzt, so dass die Induktionsannahme nicht verletzt wird.) Durch Induktion folgt daher, dass sich unter den k ersten Behältern keine leeren befinden können.

Zur Analyse der Laufzeit:

Beim ersten Gegenstand gibt es nur eine Möglichkeit, beim zweiten zwei Möglichkeiten (lege den Gegenstand in den Behälter 1 oder in den Behälter 2), beim dritten drei usw. Ab dem m-ten Gegenstand hat man für jeden Gegenstand nur noch höchstens m Möglichkeiten. Somit werden im ungünstigsten Fall

$$1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot m \cdot m \cdot ... \cdot m = m! \cdot m^{(n-m)}$$

Möglichkeiten durchprobiert (m! ist die Fakultätsfunktion). Wir müssen also größenordnungsmäßig mit m^{n-1}/e^m Schritten (wegen der Stirlingschen Formel für die Fakultät) rechnen. Dies war zu erwarten, da BPP, wie gezeigt, mindestens so aufwändig wie das spezielle Rucksackproblem ist, dessen Backtracking-Bearbeitung bereits 2^n Schritte benötigt. Für die Praxis ist das Verfahren BTBPP also nur für wenige Gegenstände (n sollte kleiner als 30 sein) einsetzbar.

Zur Analyse des Speicherplatzes:

Zusätzlich zu den globalen Variablen wird weiterer Speicherplatz nur durch die Rekursion erforderlich. Da die Rekursionstiefe, also die maximale Zahl ineinander geschachtelter Prozeduraufrufe, höchstens n ist (denn dann erfolgt der Abbruch durch "if j=n ..."), brauchen wir also zusätzlich nur "linear viel Speicherplatz", d.h., zusätzliche $c \cdot n$ Speicherplätze für eine geeignete Konstante c.

Diese Konstante hängt auch von der Implementierung der Rekursion in der jeweiligen Programmiersprache ab, so dass sie nicht ohne eine konkrete Implementierung angegeben werden kann. Dennoch kann man grob geschätzt sagen, dass man etwa den gleichen Speicherplatz, der für die Eingabe erforderlich ist, nochmals benötigt. Dies ist aber bei heutigen Rechenanlagen meist unproblematisch.

Die Analyse von Speicherplatz und Laufzeit zeigt: Für praktisch relevante Aufgabenstellungen ist das Backtracking ungeeignet. Wir brauchen also schnellere Verfahren.

Wir kennen zur Zeit keine Verfahren, die eine Lösung des Binpackings, sofern sie existiert, zuverlässig finden und zugleich größenordnungsmäßig weniger Zeit als d^n Schritte (für eine Konstante d) benötigen; man vermutet, dass es solche Verfahren nicht gibt. Daher hat man Näherungsverfahren entwickelt, die schnell sind und häufig eine Lösung, sofern sie existiert, finden; wenn diese Verfahren aber keine Lösung entdecken, so bedeutet dies nicht, dass es nicht doch eine Lösung geben kann.

Wie sehen solche Näherungsverfahren aus? Vermutlich haben Sie eines schon unbewusst benutzt, als Sie Lösungen zu den vorgestellten Beispielen gesucht haben: Man sortiere zunächst die Gegenstände nach ihrer Größe (beginnen dmit dem größten) und platziere den jeweils nächsten Gegenstand in den Behälter mit der kleinsten Nummer, in den er noch passt. Auf Näherungs-verfahren werden wir hier aber nicht eingehen.

Als nächstes wollen wir das Füllproblem, das wir anfangs behandelt haben, mit Hilfe eines Backtracking-Verfahrens lösen.

Dieses Problem ist schwieriger als das Rucksackproblem, da wir für die Rekursion nicht einfach zum folgenden Gegenstand, also zu einer nächsten Position übergehen können; denn was ist auf einer Fläche "die nächste Position"? Dies werden wir als erstes klären müssen.

5.6 Ein Algorithmus für das Füllproblem

Auch das Füllproblem wollen wir durch systematisches Probieren lösen. Wegen der vielen Nebenbedingungen ist dies aber deutlich schwerer als das Rucksack- oder das Erbschaftsproblem.

In diesem Abschnitt kommt es vor allem auf zweierlei an:
- hohe Exaktheit bei schrittweisem Vorgehen,
- Durchhaltevermögen.

Mit diesem Abschnitt können sich alle Leserinnen und Leser "testen", ob sie die für die Programmentwicklung notwenige Disziplin bei gleichzeitiger Aufmerksamkeit mitbringen. Der Abschnitt ist nicht leicht zu bearbeiten und Sie sollten sich dafür Sorge tragen, beim Durcharbeiten möglichst nicht gestört zu werden. Der Abschnitt 5.6 ist wegen der vielen Details auch recht umfangreich.

5.6.1 Vorüberlegungen

Das Füllproblem lässt sich durch systematisches Probieren lösen. Bisher funktionierte das systematische Probieren so, dass jeweils das nächste Element der Folge entweder hinzugenommen wurde oder nicht (Rucksackproblem) oder indem es in einen der vorhandenen Behälter oder in den nächsten leeren Behälter gelegt wurde (Binpacking).

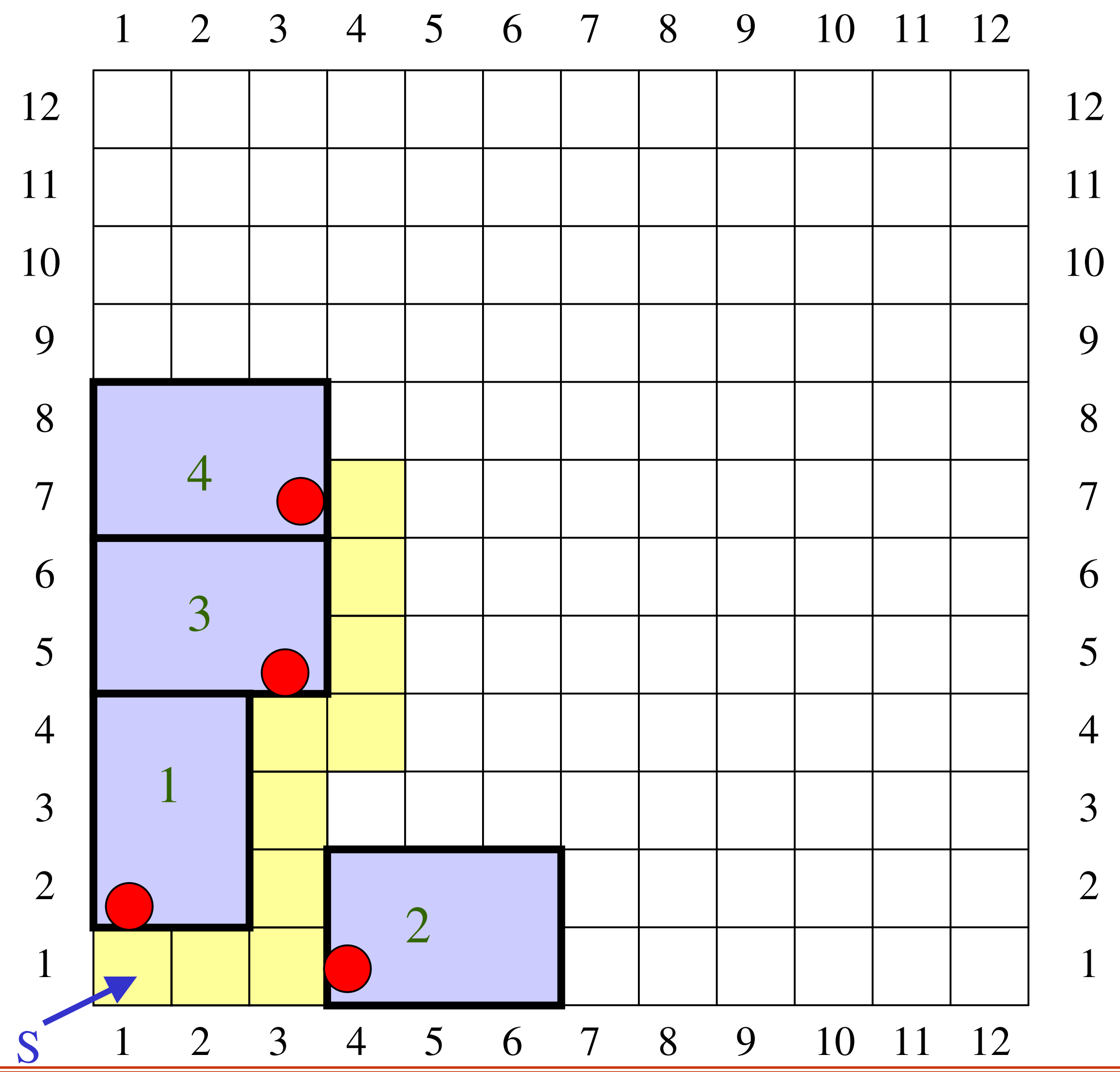
Dieses Vorgehen setzt Anordnungen voraus: Die Gegenstände und die Behälter sind durch ihre Nummern angeordnet. Beim Füllproblem müssen wir diese Anordnung erst noch festlegen, da sie a priori nicht vorhanden ist. Dies wird dazu führen, dass wir häufig arrays statt Mengen verwenden müssen.

Wir untersuchen hierfür zunächst das Beispiel (n=12, m=3, zwei Muster) aus Abschnitt 5.1 zum Platzieren von Häusern auf dem Baugrundstück.

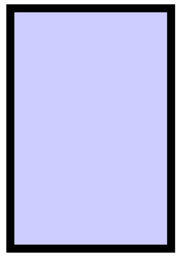
Wir gehen wie in Abschnitt 5.5 vor: Wir betrachten eine beliebige Situation und legen dann fest, wie der nächste Schritt (der nächste Prozeduraufruf) erfolgen soll. Hierzu sehen wir uns zunächst ein Beispiel an:

Wir sind bei dieser Situation angekommen, wobei wir die Häuser in der Reihenfolge der angegebenen Nummerierung 1 bis 4 eingefügt und Wege nach S festgelegt haben.

Wo fügen wir nun das 5. Haus ein?



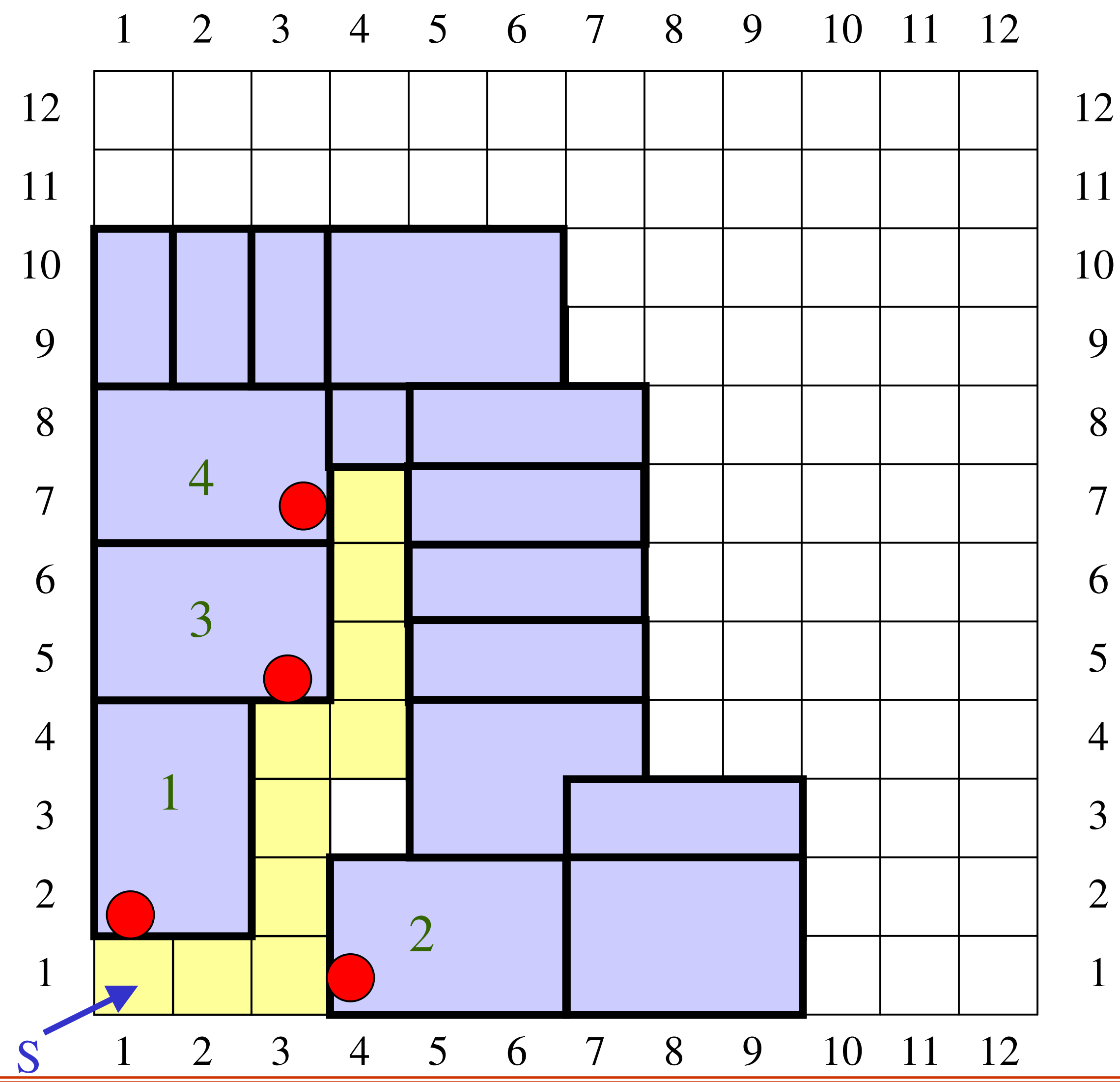
Es liegen in unserem Beispiel aus Abschnitt 5.1 zwei Muster vor, nämlich ein waagerechtes und ein senkrechtes Rechteck der Größe zwei mal drei.

Wir legen nun für jedes der Muster  und  fest,

an welchen Positionen sie im nächsten Schritt eingefügt werden dürfen. Dabei wollen wir sie so platzieren, dass sie möglichst nahe an bereits bestehenden Häusern liegen. Im Beispiel ist dies einigermaßen einfach, indem man alle Positionen, in denen das nächste Haus stehen darf, auflistet.

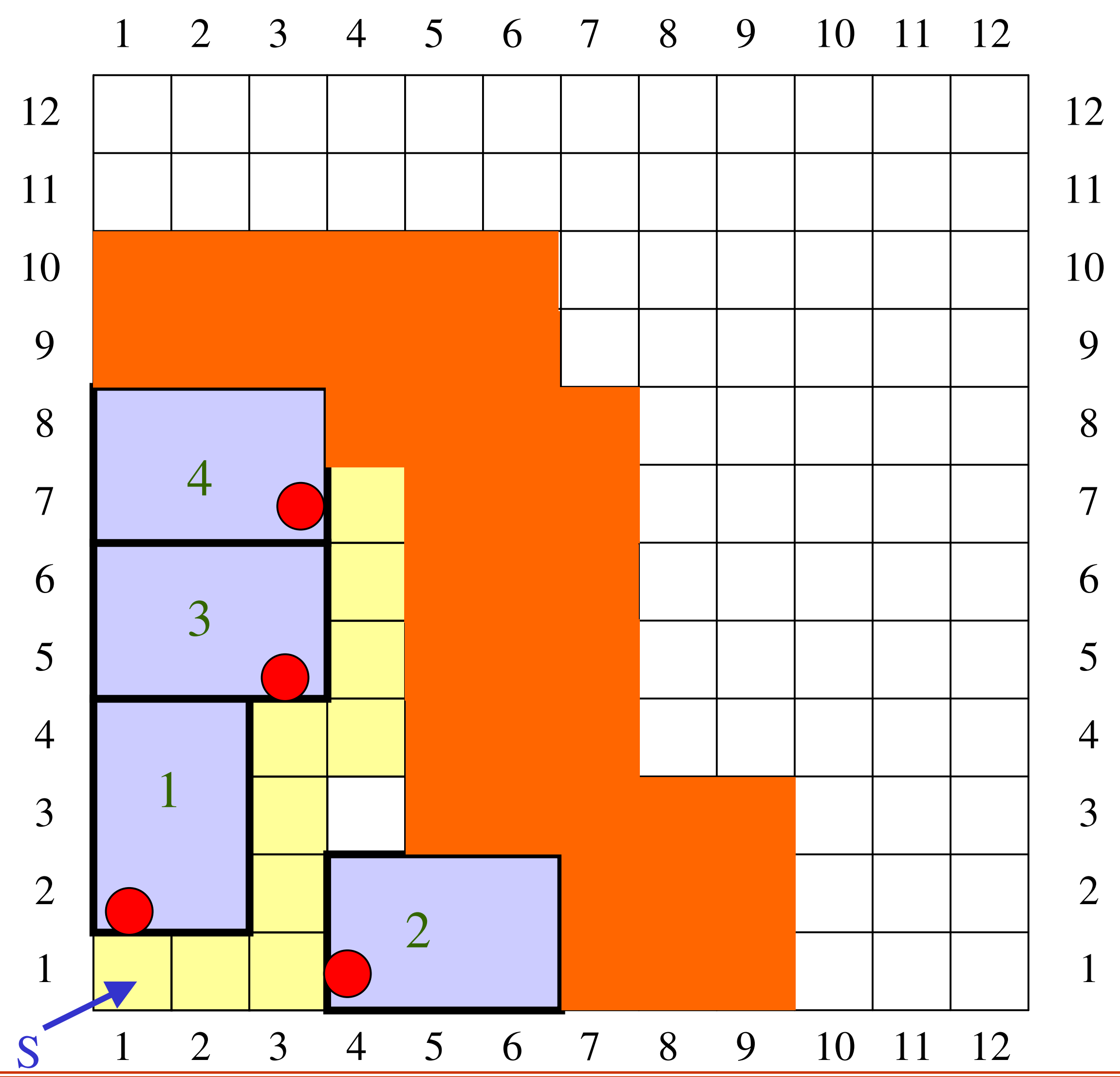
Wo können diese Positionen liegen?

Für das Muster  kommen z.B. folgende Positionen in Frage:

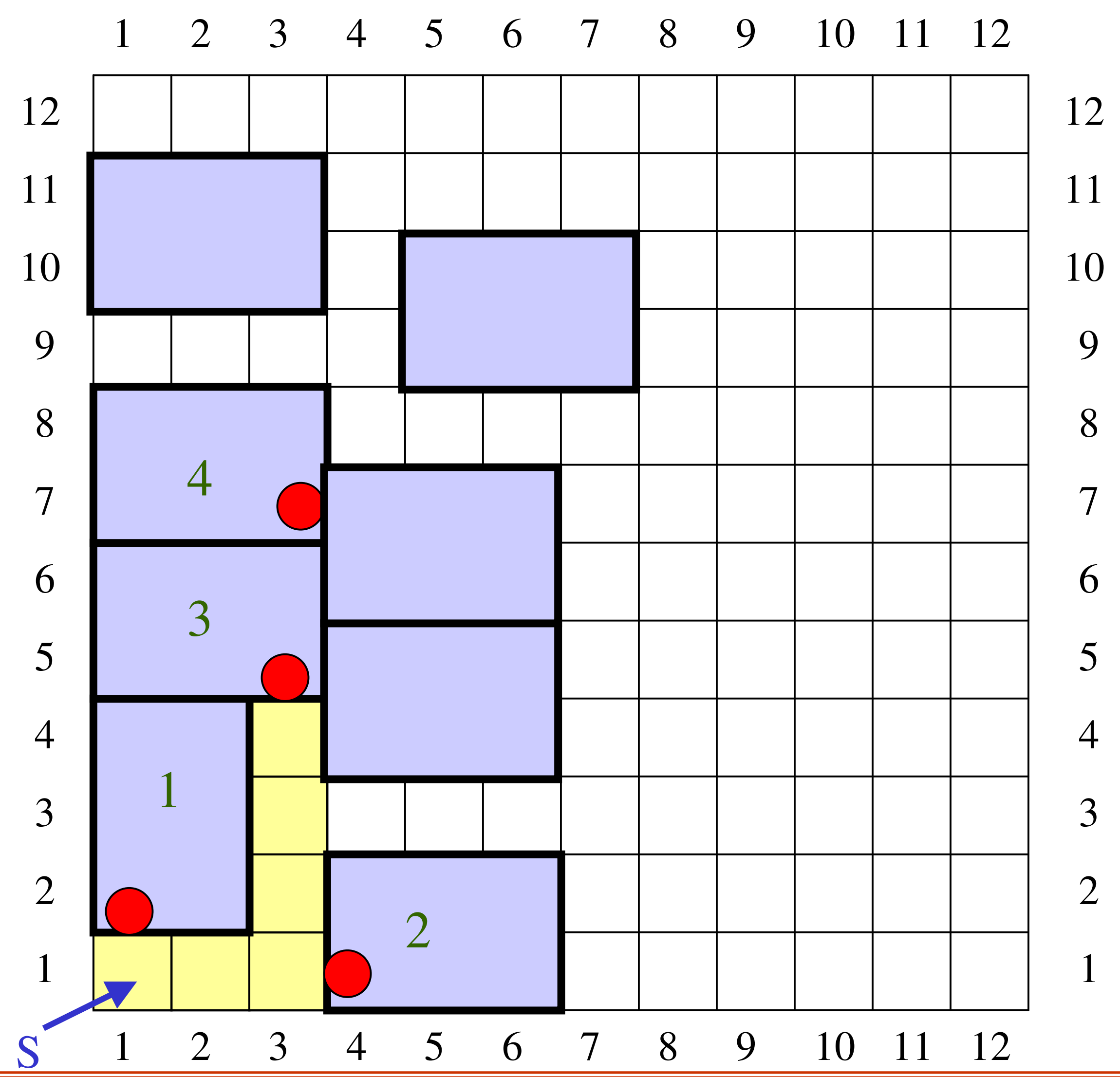


Wo fügen wir das 5. Haus ein?

Das Muster  kann also irgendwo in der roten Fläche liegen:



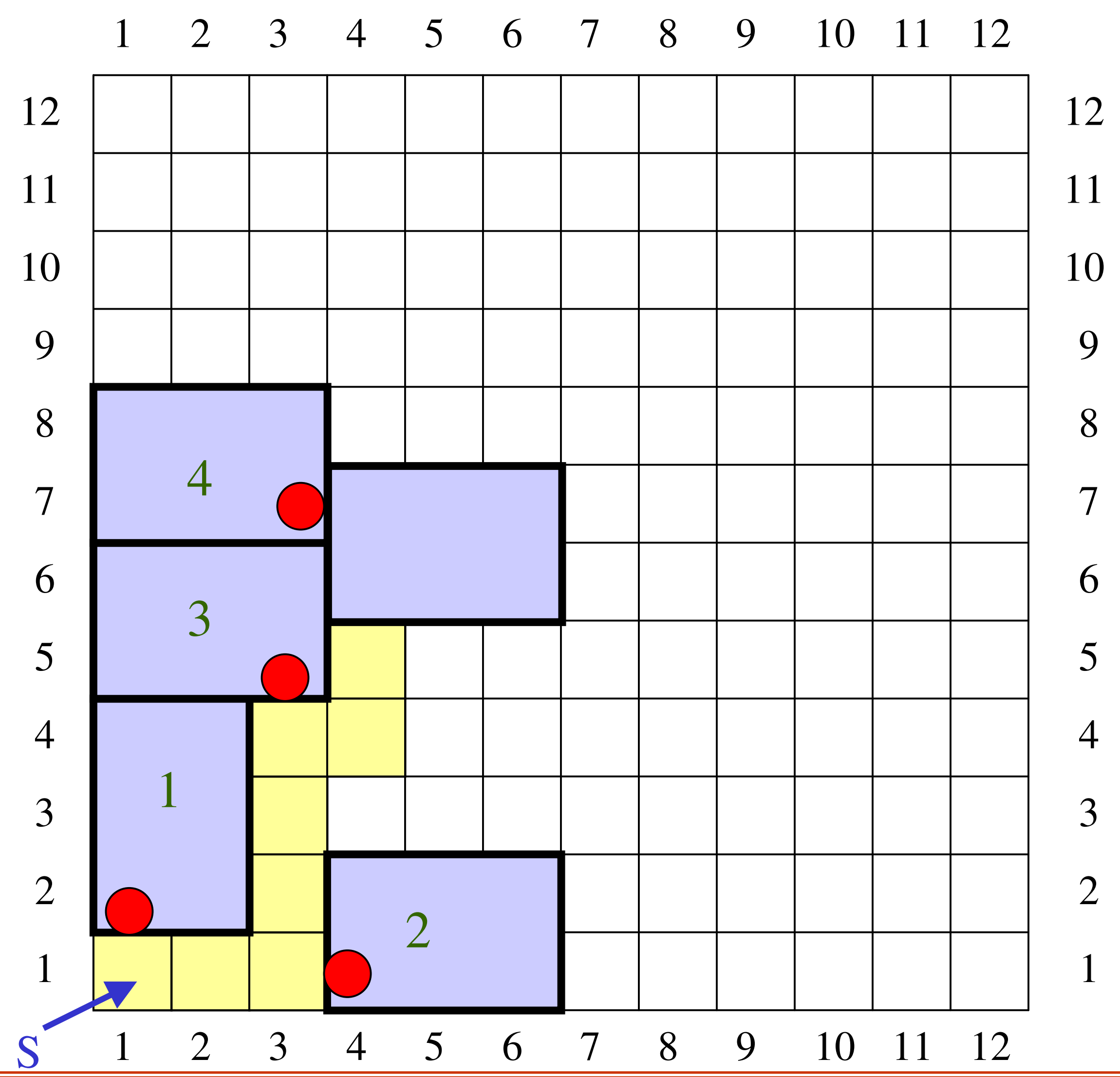
Doch diese Fläche reicht noch nicht aus. Möglich sind auch Positionen wie:



Wo fügen wir das 5. Haus ein?

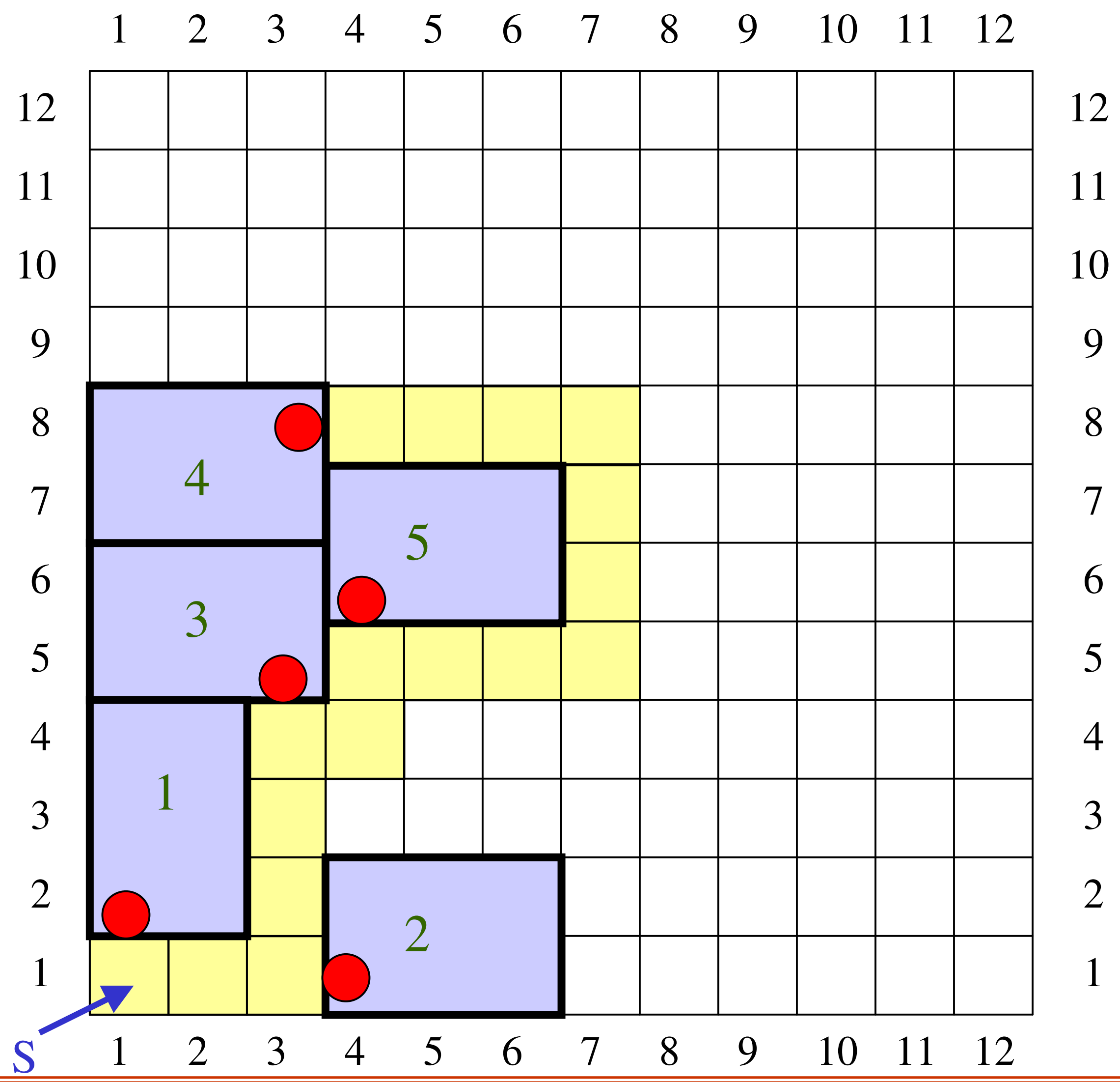
Greifen wir einmal eine dieser Situationen heraus, z.B.:

Es sieht aus, als wäre diese Lage unsinnig, da die Wege neu verlegt werden müssen. Sie kann sich aber später als gut geeignet erweisen, so dass man sie nicht einfach weglassen darf.

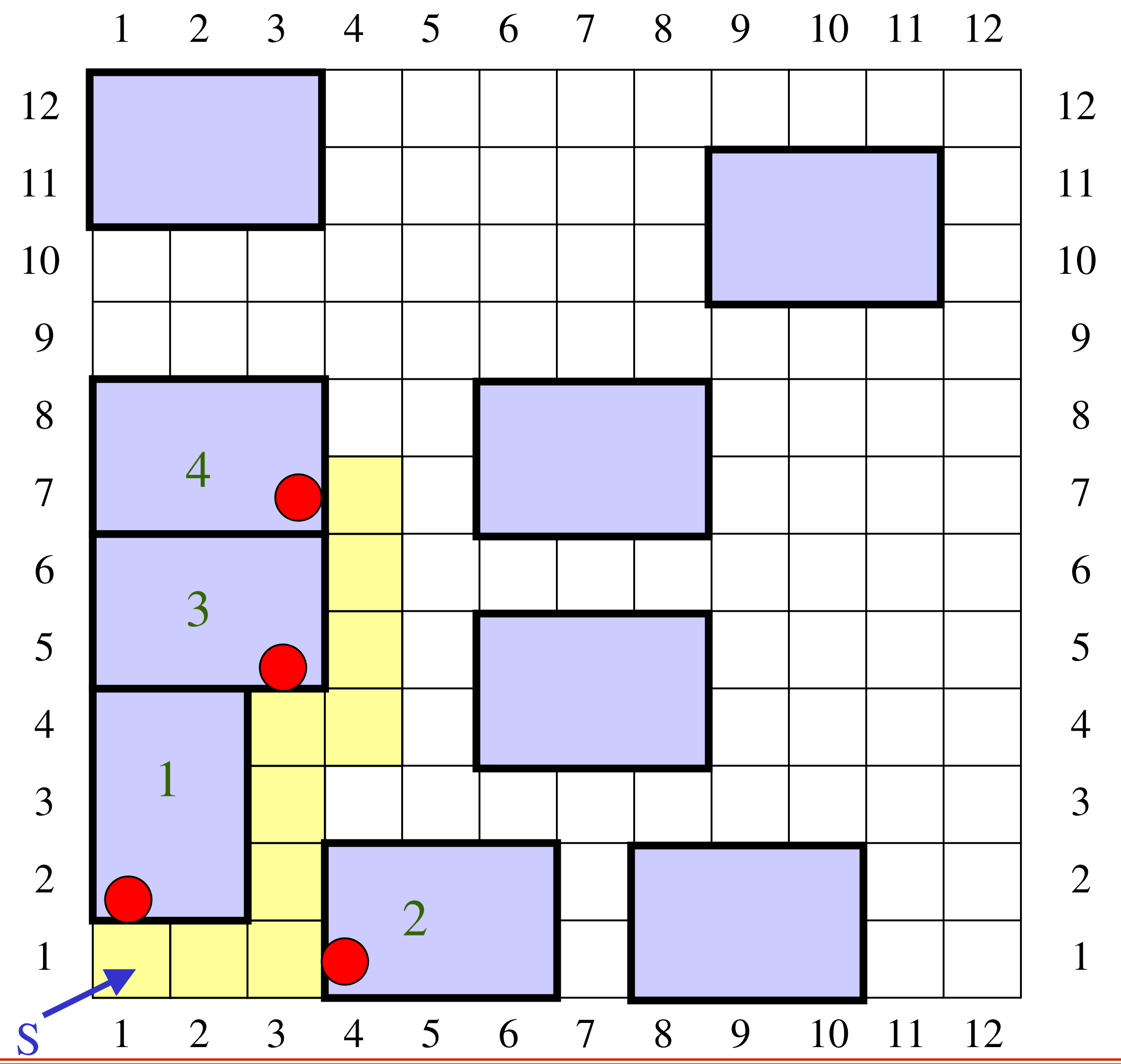


Nach dieser Platzierung ergibt sich folgende Situation:

Man sieht hieran, dass beim Platzieren und später auch beim Rückgängigmachen stets Anpassungen der bisherigen Situation erfolgen müssen.



Sind diese Platzierungen bereits alle Möglichkeiten? Auch folgende und viele weitere Positionen sind denkbar, selbst wenn man vermuten wird, dass manche von ihnen nicht zu optimalen Lösungen führen werden:

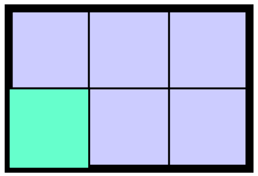


Wo fügen wir das 5. Haus ein?

Was ist die "einfachste" Konsequenz? Wir ziehen *alle* Positionen als mögliche Einfügestellen für das nächste Haus in Betracht; wir müssen dann nur testen, dass

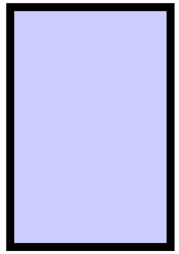
- das eingefügte Haus nicht über die Grundstücksgrenze hinausragt,
- sich keine Häuser überlappen,
- das Startfeld nicht überdeckt wird und
- stets noch ein Weg von jedem Haus zum Startfeld möglich ist.

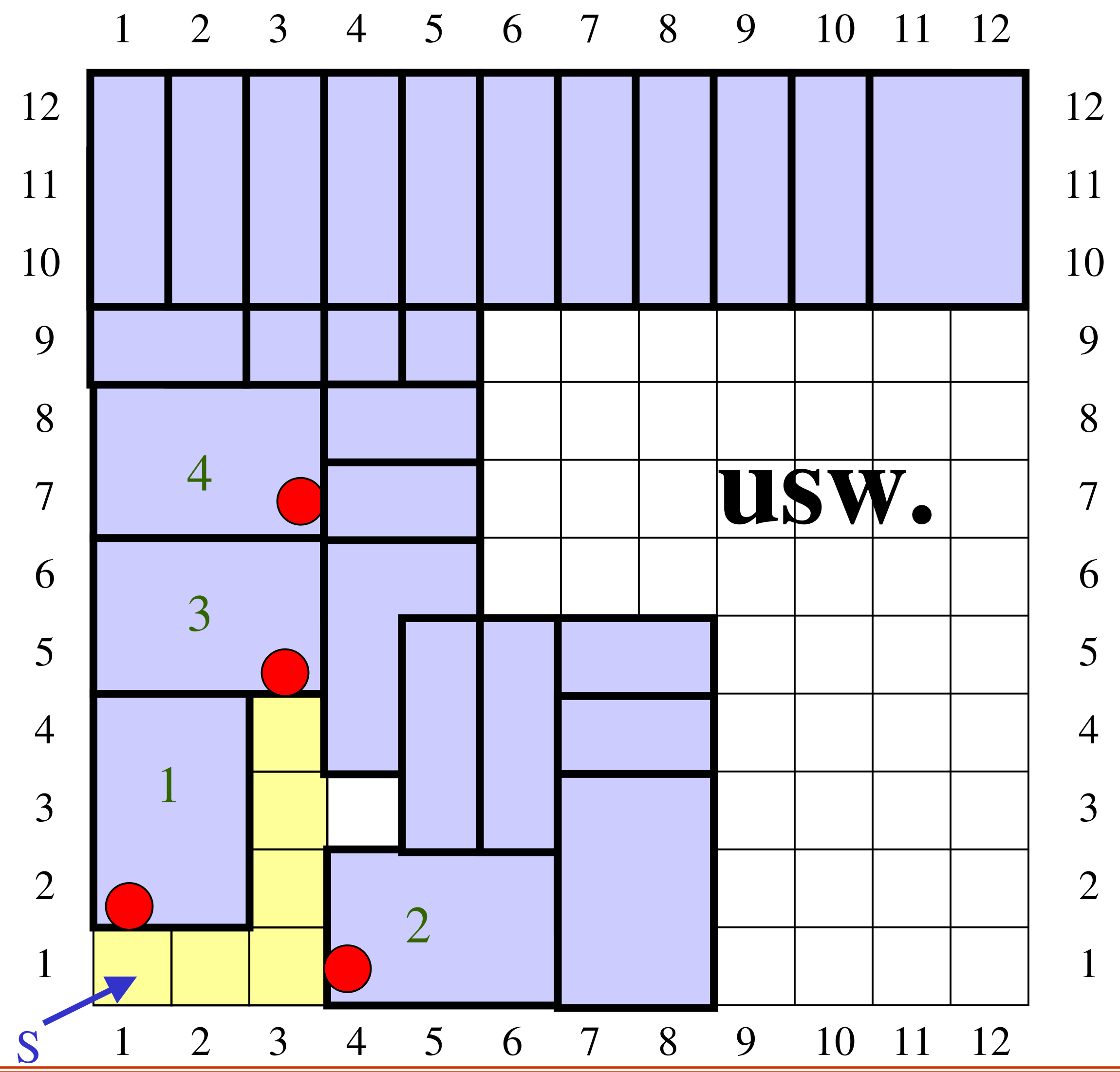
Was sind *alle* Positionen? Hierzu zeichnen wir eines der Felder des Musters aus, z.B. das Feld unten links (wir nennen es das "Leitfeld"),



und legen dieses Feld der Reihe nach auf die Positionen (1,1), (1,2), (1,3), ..., (1,n), (2,1), (2,2), (2,3), ..., (2,n), (3,1), (3,2), ..., (3,n), ..., (n,1), (n,2), ...(n,n), wobei wir jedes Mal die oben genannten vier Bedingungen testen.

Falls alle vier Bedingungen erfüllt sind, rufen wir die Prozedur rekursiv für das nächste Haus auf, sonst beenden wir den aktuellen Prozeduraufruf.

Das Gleiche müssen wir natürlich auch für das Muster  tun:



Wo fügen wir nun das 5. Haus ein?

5.6.2 Datentypen, Variablen

Das Beispiel hat uns gezeigt, wie wir vorzugehen haben: Wir versuchen in einer Situation, das nächste Muster konfliktfrei auf dem Baugrundstück unterzubringen. Gelingt dies, so fahren wir rekursiv fort; anderenfalls müssen wir die Situation genau analysieren, denn es könnte eine Lösung vorgelegen haben.

Wie aber sollen wir vorgehen, um keine der möglichen Lösungen auszulassen? Wir wählen hierfür den einfachsten Ansatz.

Die einfachste Vorgehensweise:

Lege in jedem Muster ein Feld, das "Leitfeld", fest.

Lege eine Reihenfolge der Felder des Baugrundstücks fest, z.B. (1,1), (1,2), (1,3), ..., (1,n), (2,1), (2,2), ..., (2,n), (3,1), ..., (3,n), ..., (n,1), ..., (n,n).

Für jedes Muster lege man das Leitfeld auf jede Position des Baugrundstücks in der festgelegten Reihenfolge. Für jede der hierbei entstehenden Situationen teste man, ob die vier Bedingungen erfüllt sind.

Falls ja, notiert man die Information über die entstandene Situation in globalen Variablen, führt den rekursiven Prozeduraufruf durch und macht die Information nach Beendigung des Pozeduraufrufs wieder rückgängig.

Falls aber kein weiteres Haus mehr platziert werden konnte, hat man eine Lösung gefunden, deren Qualität (Zahl der Häuser H und mittlerer Abstand μ) man ermittelt und ggf. speichert oder ausgibt.

Die jeweilige Situation muss man sich dabei in globalen Variablen merken.

Vielleicht geht es Ihnen, liebe Leserinnen und Leser, wie uns, dass Sie nun gerne "los-programmieren" möchten. Aber das sollten Sie (noch) nicht tun. Zunächst müssen wir genau klären, was die grundlegenden Daten und ihre Datentypen sind.

Diese schreiben wir jetzt auf. Dabei ist natürlich klar, dass wir auf die in Abschnitt 5.2 aufgeführten Definitionen zurückgreifen müssen, wenn wir beispielsweise die Begriffe "zulässiges Muster" oder "Haus" in einer Programmiersprache formulieren wollen. Wir werden hierfür die Definitionen 5.2 und 5.3 wiederholen und die anderen Definitionen implizit verwenden. Lesen Sie bitte nochmals die damaligen Definitionen durch.

Nun also zu den Datentypen und den zu verwendenden Variablen:

Es möge AnzPos ("Anzahl der Positionen") viele Positionen geben, auf die das Leitfeld eines jeden Musters gelegt werden kann. Eine "Position" ist das x-y-Paar eines Feldes, also vom Typ <u>array</u> [1..2] <u>of</u> natural.

S sei das "Startfeld".

H sei die Anzahl der bisher platzierten Häuser, Hmax sei die maximale Anzahl an Häusern, die bisher in einer Lösung gefunden wurde.

Die zulässigen Muster bilden eine Menge "ZulMust". Jedes Element von ZulMust ist ein Muster, d.h., eine Menge von Feldern, unter denen zusätzlich ein "Leitfeld" ausgezeichnet ist, das später vom Algorithmus auf die jeweilige Position gelegt wird. Aus einer Position und einem Muster erhält man eindeutig ein "haus", d.h. die Felder, die zu einem Haus auf dem Gesamtgrundstück gehören.

AnzMust sei die Anzahl der vorgegebenen Muster.

In den folgenden Algorithmen werden wir immer wieder Mengen durchlaufen müssen. Hierfür eignet sich die Datenstruktur "set of ..." wenig. Daher werden wir ZulMust als array anlegen, d.h., ZulMust wird eine Variable vom Typ <u>array</u> [1..AnzMust] <u>of</u> Muster sein.

"Wege" sei die Menge der Felder, über die man vom Startfeld S zu jedem bisher platzierten Haus gelangen kann. Wir formulieren dies zunächst als Menge, werden aber in den späteren Algorithmen vermutlich eine andere Datenstruktur wählen.

Mue sei der mittlere Abstand aller Häuser zum Startfeld in der aktuellen Situation. MueMin[h] sei der minimale mittlere Abstand, der bisher in einer Lösung mit h Häusern gefunden wurde.

Wir können also bereits folgendes ausformulieren:

type feld = array [1..2] of natural;   < Wir verwenden natural statt [1..n], um später keine Konflikte zu erhalten, wenn wir zu einer Komponente eine Zahl addieren wollen. >

Variablen:
AnzPos, H, Hmax, AnzMust: natural;
positionen: array [1..anzpos] of feld;
S: feld;
ZulMust: array [1..AnzMust] of Muster;   < ZulMust enthält AnzMust Elemente >
Wege: set of feld;   < Die Wege müssen eine Nebenbedingung erfüllen! >
Mue: real;
MueMin: array [1..Hmax] of real;

Ungeklärt bleiben noch die Begriffe

Muster bzw. Zulässiges Muster
Situation (Belegung mit Häusern und Wegen)
Loesung

denen wir uns auf den folgenden Folien zuwenden.

5.6.3 Muster und der Test

Ein systematisches Probierverfahren wird alle Muster auf allen Positionen durchprobieren.

Was ist ein Muster? Wie formulieren wir es als Datenstruktur? Wie testen wir ggf., ob eine Ansammlung von Feldern ein Muster ist oder nicht?

Hierfür greifen wir auf die Definitionen aus Abschnitt 5.2 zurück.

Wiederholung der Definition 5.2:

Es sei n eine natürliche Zahl, $n \geq 1$.

a) Ein <u>Muster</u> z ist eine nicht-leere Menge von Feldern

$z = \{(i_1,j_1), (i_2,j_2), \ldots, (i_m,j_m) \mid m \geq 1,\ 1 \leq i_r \leq n \text{ und } 1 \leq j_r \leq n \text{ für } r = 1,\ldots, m\}$

mit folgender Zusatzbedingung für $m>1$:

für alle $(i,j), (i',j') \in z$ gibt es einen Weg von (i,j) nach (i',j'), der nur über Felder verläuft, die in z liegen, d.h.:

$\forall (i,j),(i',j') \in z$ mit $(i,j) \neq (i',j')$
$\exists$ Weg $(i_1,j_1), (i_2,j_2),\ldots, (i_k,j_k)$ mit $(i,j) = (i_1,j_1)$, $(i',j') = (i_k,j_k)$
und $\forall r = 1,\ldots, k$: $(i_r,j_r) \in z$

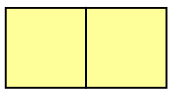
b) $m = |z|$ heißt die <u>Größe des Musters</u> z.

c) Eine Menge $Z = \{z_1, z_2, \ldots, z_s\}$ heißt <u>Menge zulässiger Muster</u>, wenn jedes z_r ein Muster gemäß Definition 5.2 a) ist und wenn alle Muster z_r die gleiche Größe besitzen.

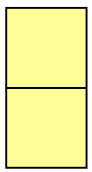
Eigentlich ist nur der Teil a) in Definition 5.2 wichtig; denn wenn unser Algorithmus *alle* Muster durchprobiert, dann ist es ihm egal, ob diese alle die gleiche Größe haben. Wir lassen daher die Forderung, dass alle Muster gleiche Größe haben müssen, weg. Wer dies zusätzlich fordern möchte, kann es leicht bei der Bildung der Menge ZulMust überprüfen.

Ein Muster ist also eine Menge von Feldern mit der Bedingung "rechteckig zusammenhängend", d.h., dass zwischen je zwei Feldern ein Weg existiert (vgl. Definition 5.1 b) und hierbei je zwei Felder der Folge "rechteckig", d.h., waagerecht oder senkrecht nebeneinander liegen müssen.

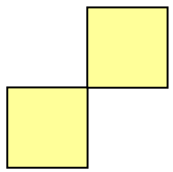
Zwei Felder (x,y) und (x',y') liegen rechteckig nebeneinander, wenn entweder x=x' und |y-y'| = 1 oder y=y' und |x-x'| = 1 gilt. Betrachte zwei Felder:



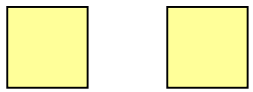
sowie



liegen rechteckig nebeneinander,



oder



liegen nicht rechteckig nebeneinander.

Ein Muster ist also eine Menge von Feldern:

<u>type</u> feld = <u>array</u> [1..2] <u>of</u> natural;
<u>type</u> muster = <u>set</u> <u>of</u> feld;   < Vorläufig! Wir ändern dies später ab. >

Weiterhin muss die Nebenbedingung

"rechteckig zusammenhängend"

erfüllt sein. Wie können wir dies für eine Menge von Feldern nachprüfen?

Wenn zwischen je zwei Feldern ein "rechteckiger" Weg existieren soll, so genügt es zu prüfen, ob von irgendeinem Feld aus alle anderen Felder durch rechteckige Schritte erreicht werden können. Dies liefert einen Algorithmus zum Testen.

Nahe liegend ist also folgendes Verfahren: Wir starten mit irgendeinem Feld (x,y) der Menge und markieren es als "besucht". Für jedes der vier rechteckig angrenzenden Nachbarfelder (x,y+1), (x+1,y), (x,y-1) und (x-1,y) prüfen wir, ob es zur Menge gehört. Falls dies der Fall ist und falls das jeweilige Feld noch nicht besucht worden war, markieren wir es als "besucht" und setzen mit diesem Feld das Verfahren rekursiv fort.

Sind am Ende alle Felder der Menge markiert, so ist sie rechteckig zusammenhängend und es handelt sich um ein Muster, anderenfalls nicht. Dies formulieren wir nun als Boolesche Funktion.

Algorithmus 6.1: Test, ob eine Menge z von Feldern ein Muster ist:

```
function BedMuster (z: muster): boolean;
var g: feld;
procedure durchsuchen (f: feld);
    begin
        for g:= (f[1],f[2]+1), (f[1]+1,f[2]), (f[1],f[2]-1), (f[1]-1,f[2]) do
            if g liegt in z and g ist noch nicht als "besucht" markiert
                then  markiere g als "besucht"; durchsuchen(g)  fi
        od
    end;
begin
    if  z ist nicht die leere Menge  then
        wähle irgendein Feld aus z und weise es g zu;
        markiere dieses Feld g als "besucht";
        durchsuchen(g);
        BedMuster := alle Felder von z sind als "besucht" markiert
    else  BedMuster := false  fi
end;
```

Um dieses Verfahren präzise in einer Programmiersprache zu formulieren, müssen wir festlegen, wie eine Menge implementiert werden soll, d.h.: Welche Datenstruktur sollen wir für die Darstellung und Verarbeitung von Mengen wählen?

Da wir die Menge z mehrmals durchlaufen müssen, sollten wir eine Ordnung auf der Menge vorgeben. Wenn die Menge m Elemente besitzt, sollten wir also anstelle von "set of .." für die Variable z den Typ

```
type  muster = array [1..m] of feld;
var  z: muster;
```

wählen. Dann können wir auch die Markierung "besucht" durch ein Boolesches array darstellen (initialisieren mit "false"):

```
besucht: array [1..m] of Boolean
```

Als "Leitfeld" des Musters z nehmen wir einfach das erste Feld in diesem array, also das Element z[1].

Nun müssen wir diese Darstellung in den Algorithmus einbauen.
Als problematisch erweist sich hierbei nur die Abfrage "g liegt in z". Hierfür müssen wir das Muster z durchlaufen und jedes Mal prüfen, ob das jeweilige Feld im Muster z gleich g ist, d.h., für z[1], z[2], ..., z[i], ..., z[m] prüfe man, ob z[i] = g ist: **<u>for</u> i:=1 <u>to</u> m <u>do</u> <u>if</u> z[i] = g <u>then</u> verlasse die Schleife <u>fi</u> <u>od</u>;**

Dies ist kein guter Programmierstil, weil man eine Schleife nicht in der Mitte, sondern nur am Ende verlassen soll (dann lässt sich ihre Korrektheit leichter einsehen oder sogar beweisen). Auch muss der Index i im Folgenden weiter benutzt werden, aber in manchen Programmiersprachen steht dieser Wert außerhalb einer Schleife nicht mehr zur Verfügung. Daher programmieren wir "sauberer" eine while-Schleife:

i:=1; <u>while</u> (i<m) <u>and</u> z[i]≠g <u>do</u> i:=i+1 <u>od</u>;

Die Abfrage "g liegt in z" ist anschließend gleichbedeutend mit "**z[i]=g**".

Machen Sie sich diese Aussage klar, indem Sie fünf Zahlen aufschreiben und diese while-Schleife daran nachvollziehen.

Fortsetzung nur per Mausclick.

Nun müssen wir noch die übrigen umgangssprachlichen Formulierungen neu fassen:

"z ist nicht die leere Menge" *ersetzen wir einfach durch* m>0,
wobei m die Anzahl der Elemente im Muster z ist.

"wähle irgendein Feld aus z und weise es g zu"
Hier können wir stets das Feld z[1] nehmen, so dass diese Anweisung entfällt.

"markiere dieses Feld g als besucht" *lautet dann einfach:* besucht[1] := true

"durchsuchen(g)" *wird zu* durchsuchen(z[1])

ergebnis := alle Felder von z sind als "besucht" markiert
muss durch eine Schleife ersetzt werden:

j:=1; <u>while</u> j<m <u>and</u> besucht[j] <u>do</u> j:=j+1 <u>od</u>;
ergebnis := besucht[j];

Nun fügen wir wegen der besseren Klarheit noch einige Klammern hinzu und erhalten aus Algorithmus 6.1:

Algorithmus 6.2: Test, ob eine Menge z von Feldern ein Muster ist:

```
function BedMuster (z: muster): boolean;
var g: feld; j, m: natural; besucht: array [1..mmax] of Boolean
procedure durchsuchen (f: feld);
    var i: natural;
    begin
        for g:= (f[1],f[2]+1), (f[1]+1,f[2]), (f[1],f[2]-1), (f[1]-1,f[2]) do
            i:=1; while  (i < m)  and  (z[i] ≠ g)  do i := i + 1 od;
            if (z[i] = g) and not besucht[i]
                then  besucht[i]:=true; durchsuchen(g)  fi
        od
    end;
begin   {ermittle zunächst m = die Anzahl der Felder in z}
    if  m > 0  then
        besucht[1] := true; for j := 2 to m do besucht[j] := false od;
        durchsuchen(z[1]);
        j:=1; while (j < m) and besucht[j] do j := j + 1 od;
        BedMuster := besucht[j]
    else  BedMuster := false  fi
end;
```

Hierbei haben wir folgende Anforderungen an die Programmiersprache:

1. Es muss möglich sein, arrays zuzuweisen und in Bedingungen zu vergleichen.
   Beispiel: g := (f[1],f[2]+1) und z[i] = g.
   Ist dies in der Sprache nicht vorgesehen, dann muss man die Anweisungen auf komponentenweise Zuweisungen umschreiben.

2. Es müssen aufzählende for-Schleifen zugelassen sein. Denn die von uns verwendete for-Schleife weist in der Prozedur "durchsuchen" dem Feld g nacheinander die vier Nachbarfelder von f zu. Ist dies nicht vorgesehen, dann kann man die for-Schleife durch vielmaliges Aufschreiben des Schleifenrumpfs ersetzen.

Schließlich muss zu jedem Muster die Anzahl m seiner Felder bekannt sein (Größe des Musters laut Definition 5.2). Hierfür kann man ein array groesse: <u>array</u> [1..AnzMust] <u>of</u> natural einführen, in dem für jedes Muster ZulMust[1], ZulMust[2], ..., ZulMust[AnzMust] dessen Größe gespeichert ist. Die globale Variable mmax ist dann das Maximum aller dieser Größen.

5.6.4 Konfigurationen

Von den drei zu klärenden Begriffen haben wir also Muster bzw. Zulässiges Muster abgearbeitet. Es verbleiben noch:

Situation (Belegung mit Häusern und Wegen)

Loesung

Wir wenden uns dem zentralen Begriff "Situation" zu. (Der Begriff "Loesung" ergibt sich "nebenbei" als eine Situation, in der man keine weiteren Häuser setzen kann.)

Eine **Situation** ist eine fotografische Momentaufnahme der Lage auf dem gesamten Baugrundstück. Eine solche Beschreibung, die zu einem Zeitpunkt alles genau erfasst, bezeichnet man in der Informatik auch als "Konfiguration". Alle Informationen können wir hierbei in einem array, das genau der Größe des Baugrundstücks entspricht, speichern:

<u>type</u> konfiguration = <u>array</u> [1..n, 1..n] <u>of</u> belegung

Mit dem Datentyp "belegung" werden alle Möglichkeiten, die für das jeweilige Feld des Baugrundstück zutreffen können, beschrieben:

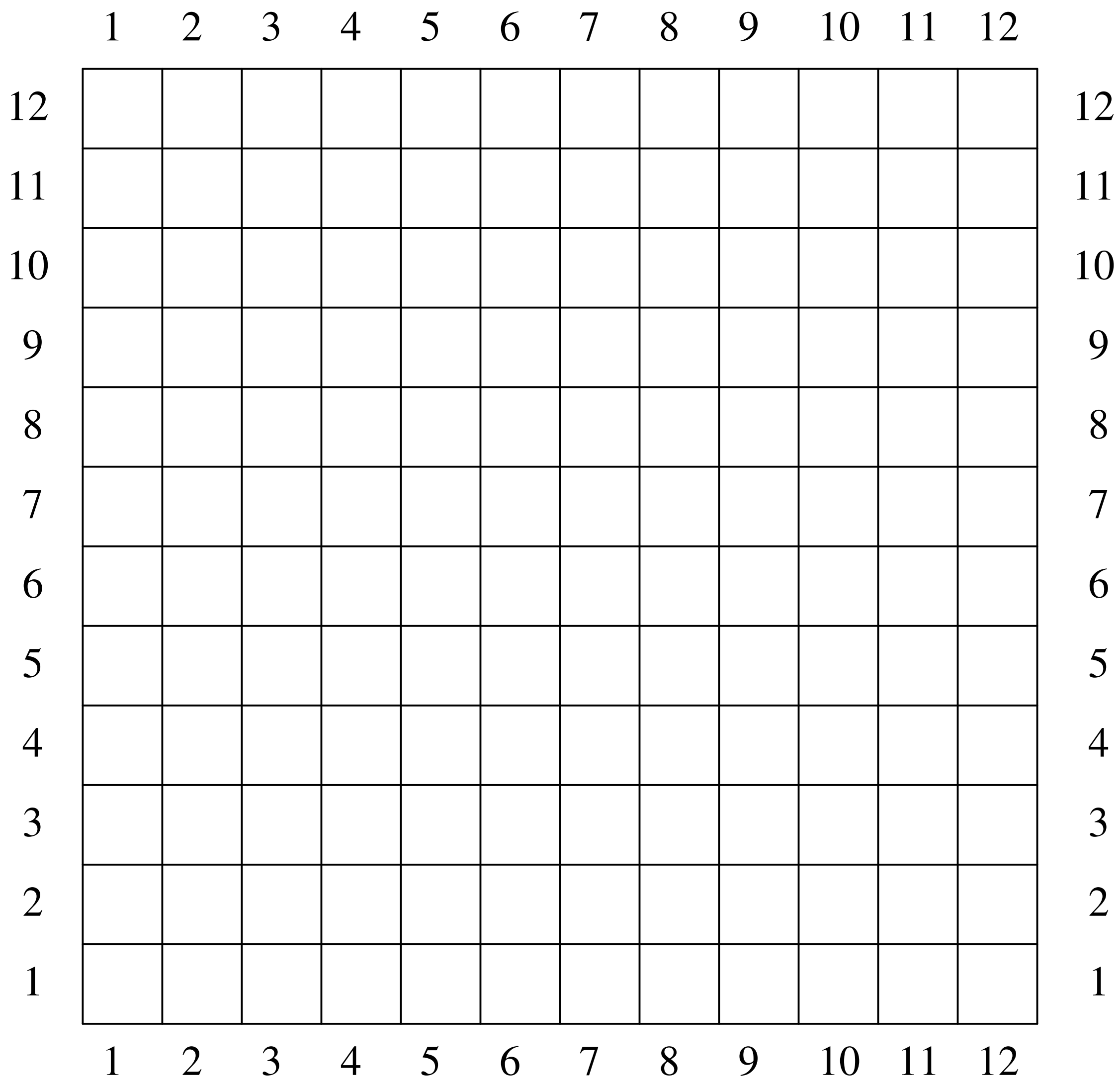
type belegung = natural union {start, frei, weg, u}

Diese Formulierung bedeutet: "Der Datentyp "belegung" ist die Vereinigung ("union") von den natürlichen Zahlen mit der endlichen Menge {start, frei, weg, u}. Jedes Feld des Baugrundstücks kann also mit einer der folgenden Möglichkeiten belegt sein:

- "start": Es ist das Startfeld.
- "frei": Es ist noch nicht verplant.
- "weg": Es gehört zu einem Weg, der zum Startfeld führt.
- "u": Es ist vom Startfeld unerreichbar.
- natürliche Zahl i: Dieses Feld gehört zum i-ten Haus.

An einem Beispiel wird dies unmittelbar klar:

Dies ist das zunächst leere Baugrundstück:



Darstellung des leeren Baugrundstücks als Konfiguration:

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun das Startfeld, z.B. (4,1) hinzu:

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	frei	frei	frei	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein Haus mit der Nummer 1 unten links hinzu:

1

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	frei	frei	frei	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein Haus mit der Nummer 1 unten links hinzu:

Neue Darstellung:

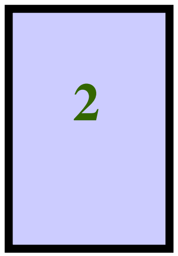
1	1	1
1	1	1

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	frei	frei	frei	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein Haus mit der Nummer 1 unten links hinzu:

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein neues Haus mit der Nummer 2 links hinzu:



	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein neues Haus mit der Nummer 2 links hinzu:

Neue Darstellung:

2	2
2	2
2	2

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Wir fügen nun ein neues Haus mit der Nummer 2 links hinzu:

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	frei	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Jetzt fügen wir einen Weg zu Haus 2 hinzu:

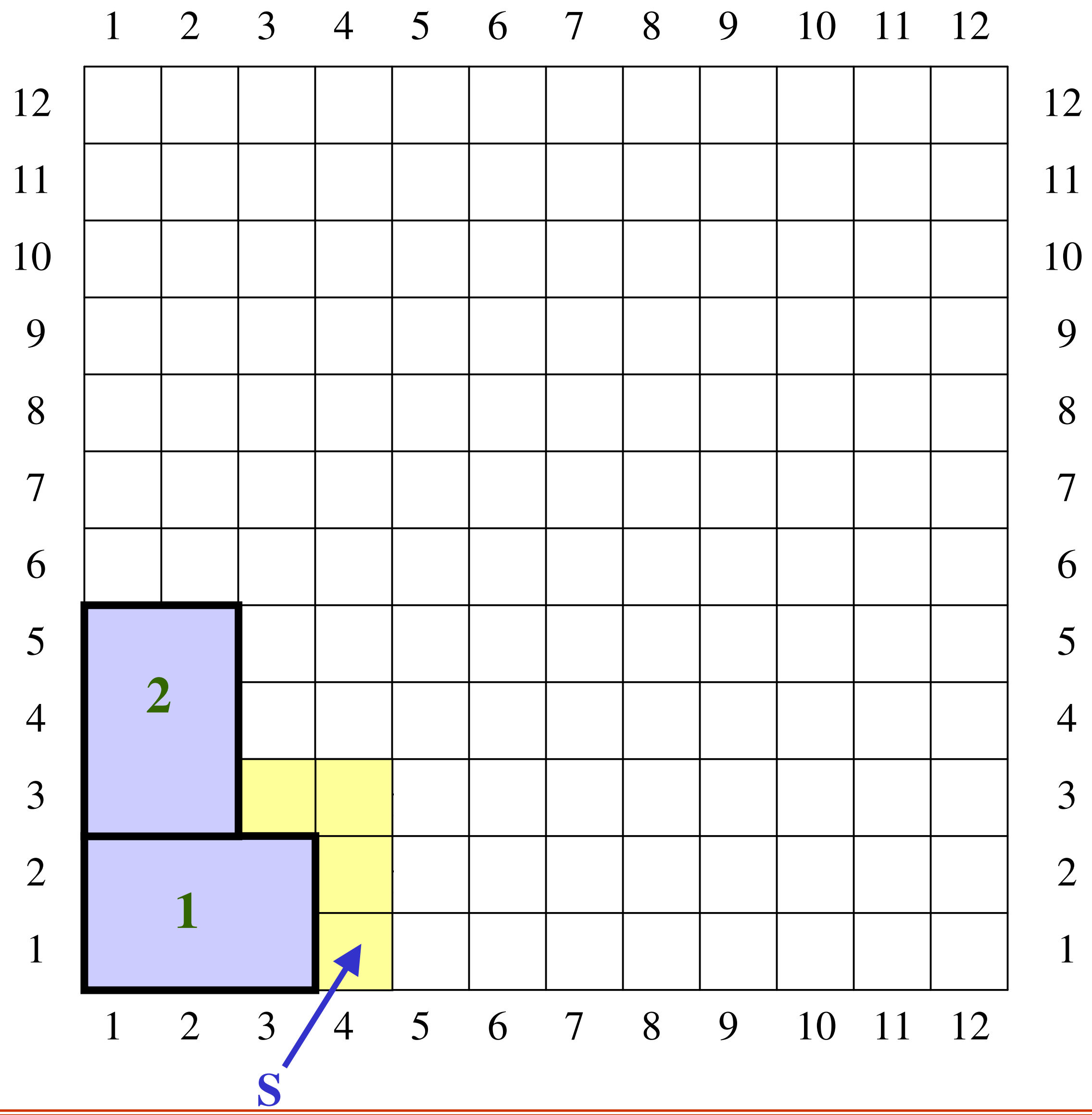
	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	2	2	weg	weg	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	weg	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

Vergleich mit unserer früheren Darstellung:

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5			frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	**2**		frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3					frei	frei	frei	frei	frei	frei	frei	frei	3
2					frei	frei	frei	frei	frei	frei	frei	frei	2
1		**1**		start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

S

In unserer früheren Darstellung entspricht dies also:



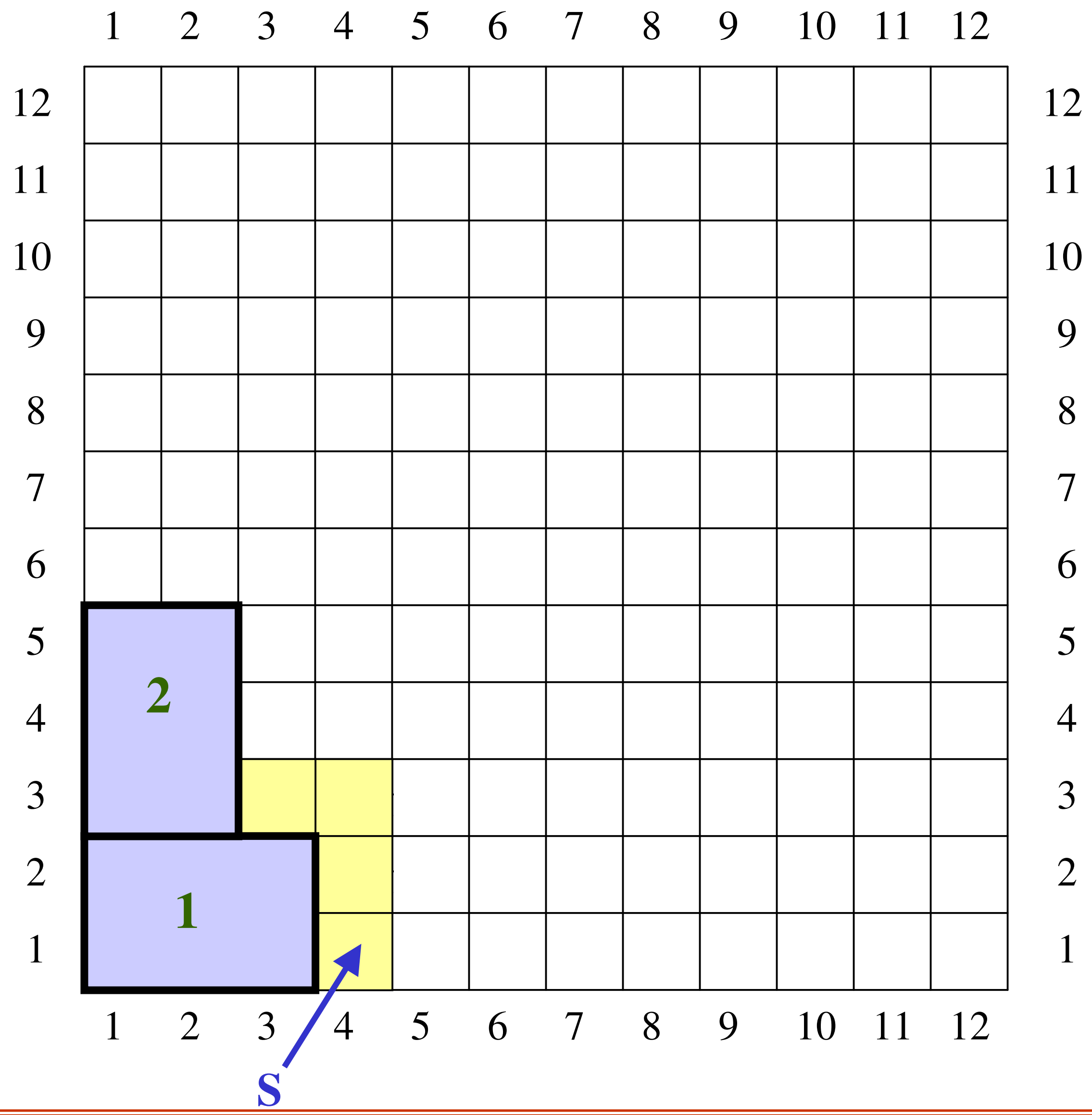
Nochmals zum genauen Vergleich:

Darstellung als Konfiguration

	1	2	3	4	5	6	7	8	9	10	11	12	
12	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	12
11	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	11
10	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	10
9	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	9
8	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	8
7	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	7
6	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	6
5	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	5
4	2	2	frei	frei	frei	frei	frei	frei	frei	frei	frei	frei	4
3	2	2	weg	weg	frei	frei	frei	frei	frei	frei	frei	frei	3
2	1	1	1	weg	frei	frei	frei	frei	frei	frei	frei	frei	2
1	1	1	1	start	frei	frei	frei	frei	frei	frei	frei	frei	1
	1	2	3	4	5	6	7	8	9	10	11	12	

S

Anschauliche Darstellung:



Wir führen nun also eine Variable "situation" ein, die zu jedem Zeitpunkt den Zustand des Baugrundstücks vollständig beschreibt:

<u>var</u> situation: konfiguration

5.6.5 Prinzipielle Lösung

Nachdem wir nun die Situationen beschreiben können, greifen wir die weitere vorne angegebene Vorgehensweise wieder auf. Zur Erinnerung:

Für jedes Muster lege man das Leitfeld auf jede Position des Baugrundstücks in der festgelegten Reihenfolge. Für jede der hierbei entstehenden Situationen teste man, ob die vier Bedingungen erfüllt sind.

Falls ja, notiert man die Information über die entstandene Situation in globalen Variablen, führt den rekursiven Prozeduraufruf durch und macht die Information nach Beendigung des Pozeduraufrufs wieder rückgängig.

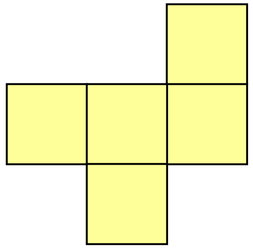
Falls aber kein weiteres Haus mehr platziert werden konnte, hat man eine Lösung gefunden, deren Qualität (Zahl der Häuser H und mittlerer Abstand μ) man ermittelt und ggf. speichert oder ausgibt.

Wir können nun bereits die Grobversion der Prozedur **BtFp** (**B**ack**t**rackingverfahren für das **F**üll**p**roblem) formulieren:

Algorithmus 6.3:

```
procedure BtFp;
begin
  for all Feld f des Baugrundstücks do
        for all z ∈ ZulMust do
              lege das Leitfeld von z auf das Feld f;
              if dies ist konfliktfrei möglich (4 Bedingungen testen!) then
                    setze Situation neu;
                    BtFp;
                    mache diese Setzung wieder rückgängig
              else evtl. lag eine Lösung vor, bewerte sie usw. fi
        od
  od
end;
```

Als erstes müssen wir das Leitfeld von z, also z[1], auf das Feld f legen. Dadurch erhalten wir ein konkretes "Haus". Als Beispiel betrachten wir folgendes Muster:



$z = \big((2,3), (3,3), (4,3), (3,2), (4,4)\big)$

mit dem Leitfeld (2,3).

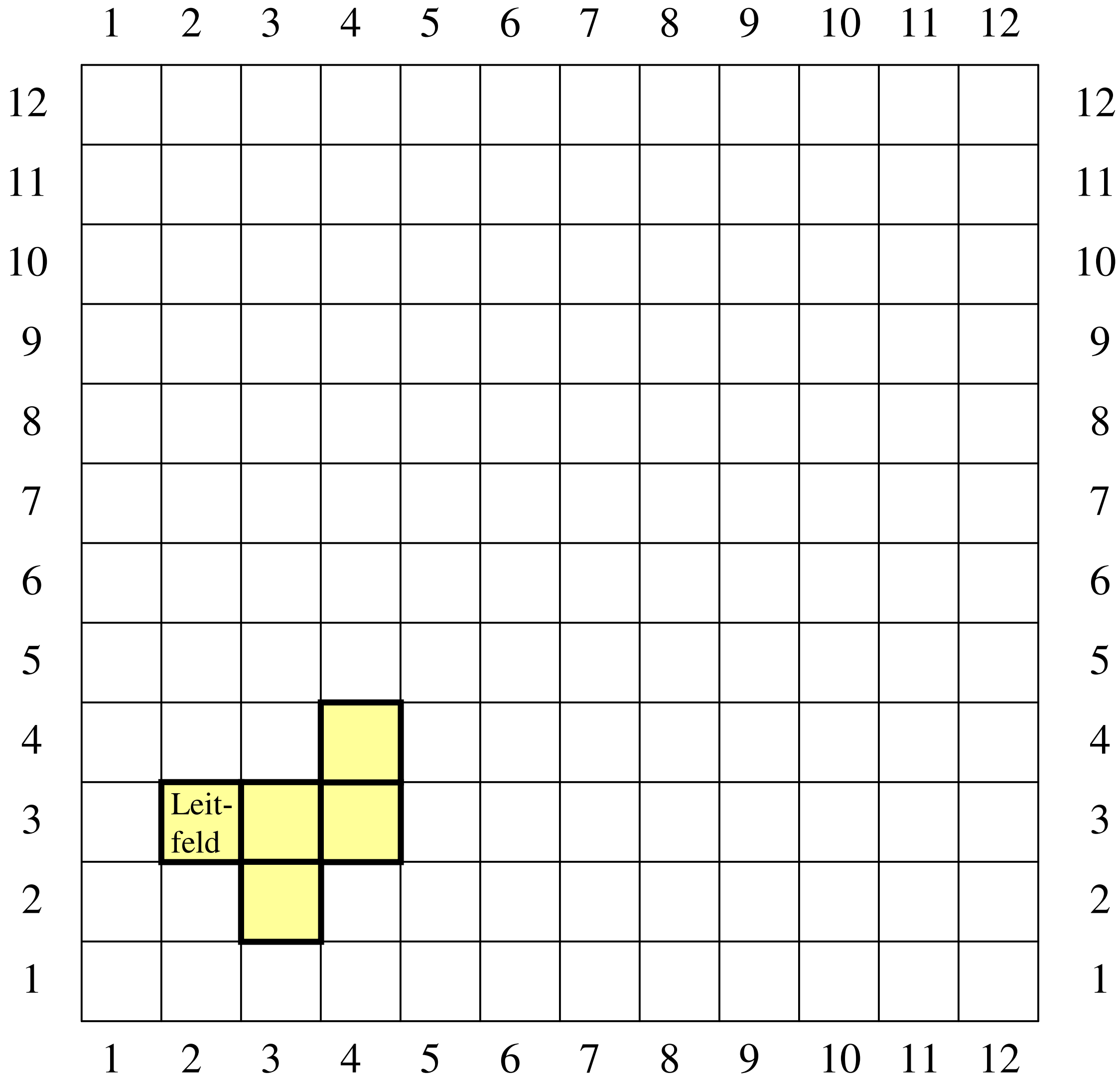
Als Feld f wählen wir willkürlich (1,1) und erhalten beim Übereinanderlegen:

$haus = \big((1,1), (2,1), (3,1), (2,0), (3,2)\big)$,

das nicht mehr vollständig im Grundstück liegt.

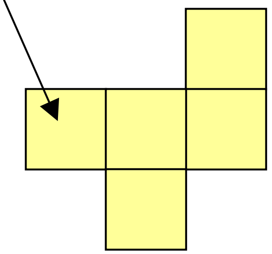
Man muss hierbei zu jedem Feld von z das Paar (-1,-2) komponentenweise addieren. Diesen Vektor (-1,-2) erhält man durch f-z[1] = (1,1)-(2,3) = (-1,-2).

Veranschaulichung des Beispiels:



Muster
(2,3),
(3,3),
(4,3),
(3,2),
(4,4).

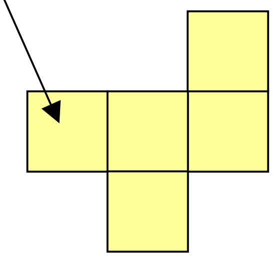
Leitfeld

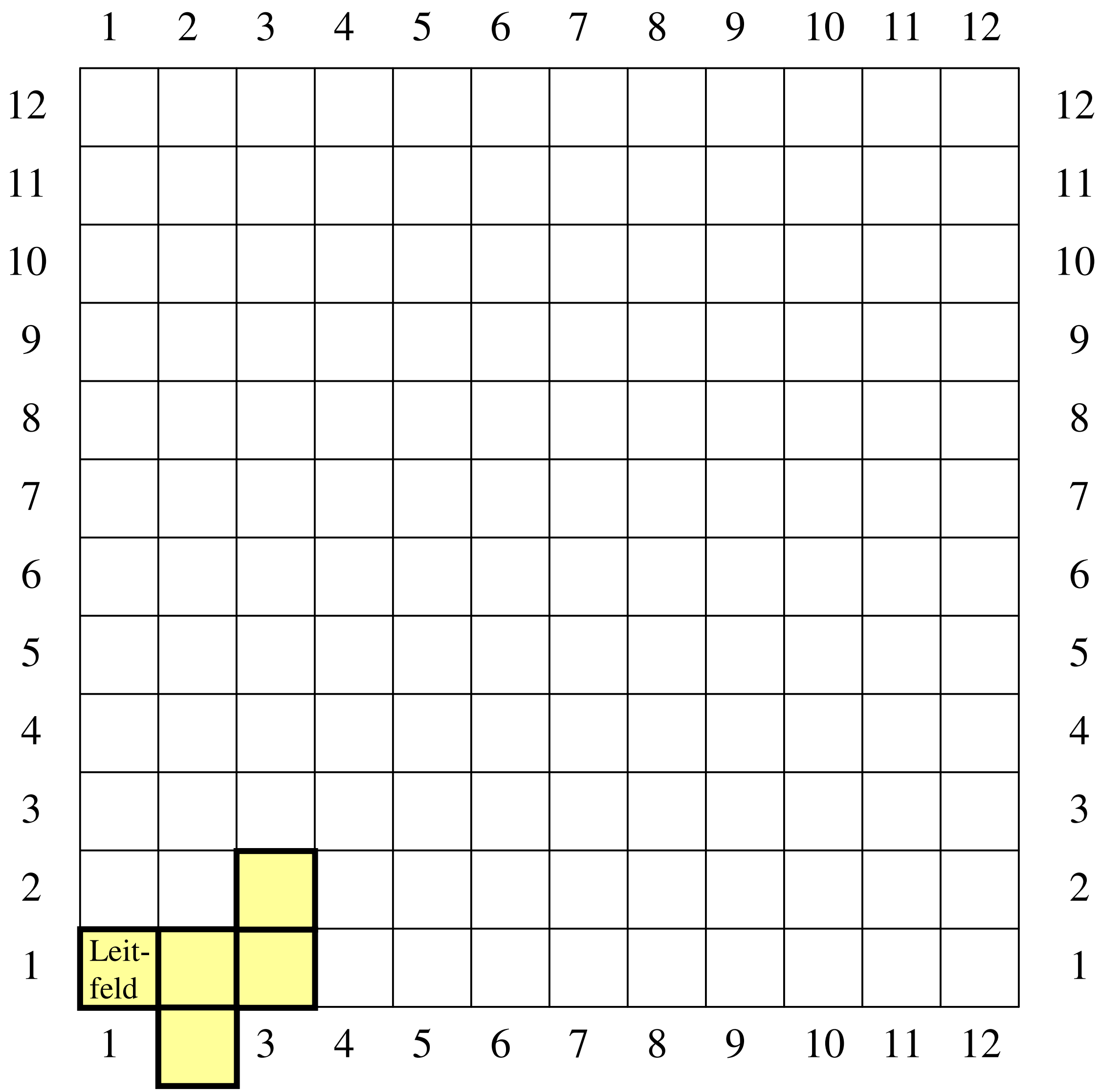


Verschiebe das Leitfeld auf das Feld (1,1).

Muster
(2,3),
(3,3),
(4,3),
(3,2),
(4,4).

Leitfeld





Dieses "haus" ist nicht zulässig.

Hieraus folgt sofort das allgemeine Vorgehen:
Aus dem Muster z, dem Leitfeld z[1] und dem Grundstücksfeld f bekommt man das konkrete Haus durch haus := z + (f-z[1]).
Diese Schreibweise besagt, dass man zu jedem Feld in z den Vektor f-z[1] addieren muss, um das konkrete "haus" zu erhalten.

Nun zurück zur Prozedur.
Als nächstes sind die vier Bedingungen zu klären, die die konfliktfreie Lage des Hauses darstellen.

Die vier Bedingungen besagten, dass
- das neu gebildete Haus nicht über die Grundstücksgrenze ragt,
- es sich mit keinem anderen Haus überlappen darf,
- es nicht das Startfeld überdeckt und
- stets noch ein Weg von jedem Haus zum Startfeld möglich ist.

Da wir genau festgelegt haben, was eine Situation ist, lässt sich für jedes neu gebildete Haus feststellen, ob diese Bedingungen erfüllt sind.

Die *erste Bedingung* lässt sich leicht nachprüfen: Es sei m die Anzahl der Felder des Musters (und damit auch des neu gebildeten Hauses); dann muss für alle i = 1, ..., m gelten
$1 \leq haus[i]\ [1] \leq n$ und $1 \leq haus[i]\ [2] \leq n$, d.h., alle Komponenten der Felder müssen zwischen 1 und n liegen.

Die *erste Bedingung* lässt sich leicht nachprüfen: Es sei m die Anzahl der Felder des Musters (und damit auch des neu gebildeten Hauses); dann muss für alle i = 1, ..., m gelten
$1 \leq$ haus[i] [1] $\leq n$ und $1 \leq$ haus[i] [2] $\leq n$, d.h., alle Komponenten der Felder müssen zwischen 1 und n liegen.

Die *zweite und die dritte Bedingung* lassen sich ebenfalls leicht testen; denn hierfür muss jedes Feld des neuen Hauses auf dem Baugrundstück entweder "frei" oder "weg" sein:
Für alle i = 1, ..., m muss also gelten:

situation [haus[i]] = frei oder situation [haus[i]] = weg .

Die *vierte Bedingung* dagegen sieht aufwändig aus. Sie lässt sich aber rasch auf ein "Kürzeste-Wege"-Problem zurückführen und mit dem Dijkstra-Algorithmus lösen.

Hierfür verwandeln wir die vorliegende Situation in einen ungerichteten Graphen: Die Felder des Baugrundstücks werden die Knoten des Graphens und zwei Knoten (x,y) und (x',y') werden genau dann durch eine ungerichtete Kante verbunden, wenn
- die Felder (x,y) und (x',y') rechteckig nebeneinander liegen und
- situation[x,y] oder situation[x',y'] keine natürliche Zahl ist.

Ein Weg darf also zu einem Haus führen, aber nicht durch Häuser hindurch, d.h., situation[x,y] und situation[x',y'] dürfen *nicht gleichzeitig* zu irgendwelchen Häusern gehören.

Dann geht man wie folgt vor: Führe ausgehend vom Startfeld S auf diesem Graphen den Dijkstra-Algorithmus zur Berechnung der kürzesten Wege zu allen anderen Feldern aus; wird hierbei für jedes Haus mindestens eines seiner Felder erreicht, so ist Bedingung vier erfüllt und der Dijkstra-Algorithmus liefert zugleich geeignete Wege von S zu jedem Haus. Markiere die Felder auf diesen Wegen mit "weg" und verwende die so erhaltene Situation als Ausgangssituation für den rekursiven Aufruf der Prozedur BtFp.

Wir führen diesen Programmteil hier nicht weiter aus. Für diejenigen, die mit Algorithmen auf Graphen vertraut sind, ist klar, dass dieses Vorgehen korrekt ist und sich implementieren lässt. Jedoch muss der Dijkstra-Algorithmus unserer Datenstruktur angepasst werden, damit nicht ständig ein Graph aufgebaut werden muss. Wir kommen hierauf bei den Implementierungshinweisen zurück.

Wir haben nun gesehen, dass und wie der Schleifenrumpf in der Prozedur BtFp als Programm ausformuliert werden kann.

Bevor wir den Algorithmus weiter entwickeln: Haben Sie den Denkfehler, der auf den letzten Folien gemacht wurde, bemerkt? Irgendetwas ist falsch, und Sie sollten diesen Fehler innerhalb von 5 Minuten finden können.

Weiter *nur per Mausclick.*

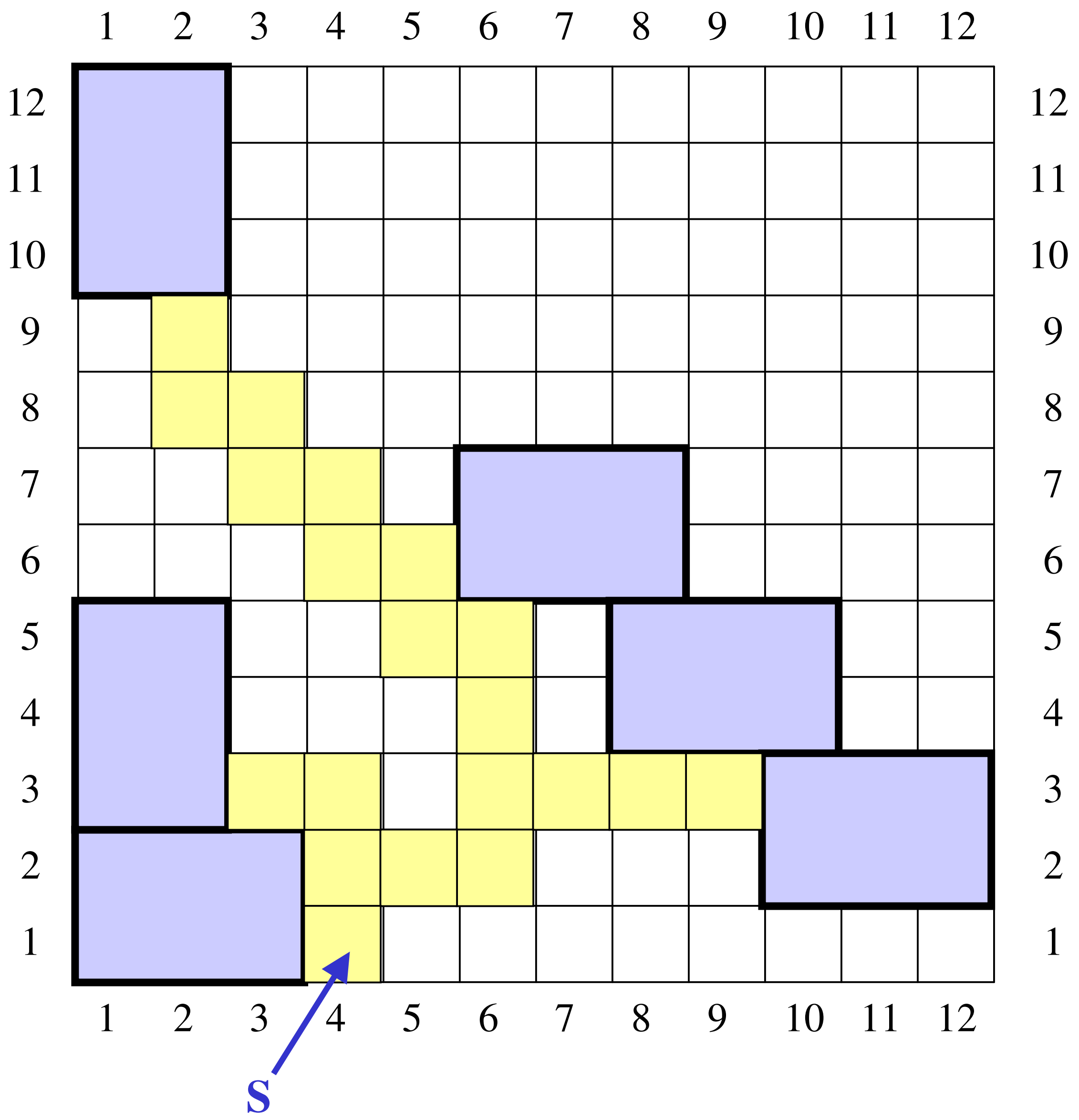
Antwort auf die obige Frage:

Die Bedingung für eine ungerichtete Kante

situation[x,y] oder situation[x',y'] ist keine natürliche Zahl

ist nicht korrekt, wie man an den folgenden Bildern sieht:

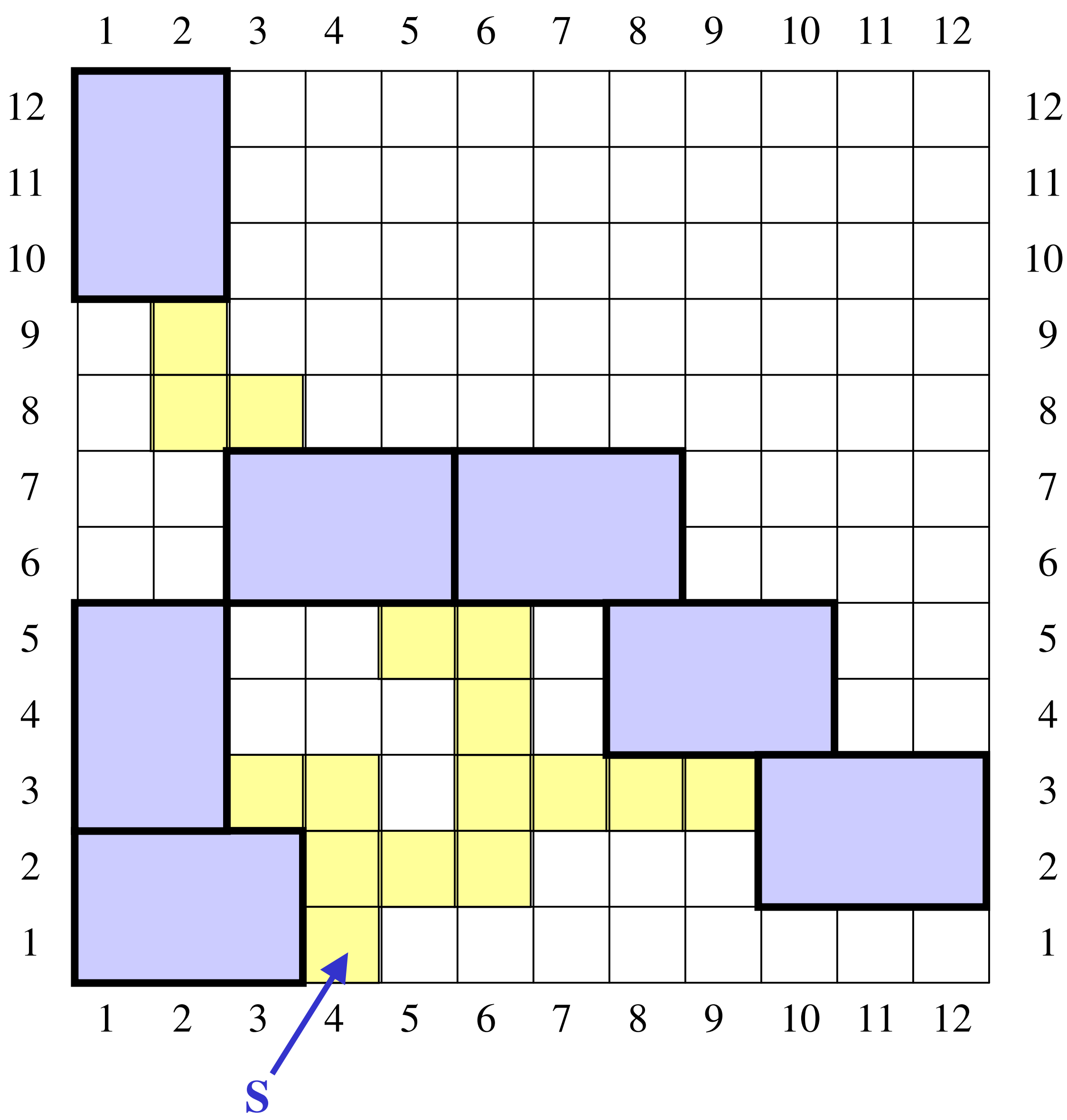
Wenn zum Beispiel folgende Situation vorliegt:



Diese Häuser mögen bereits platziert sein.

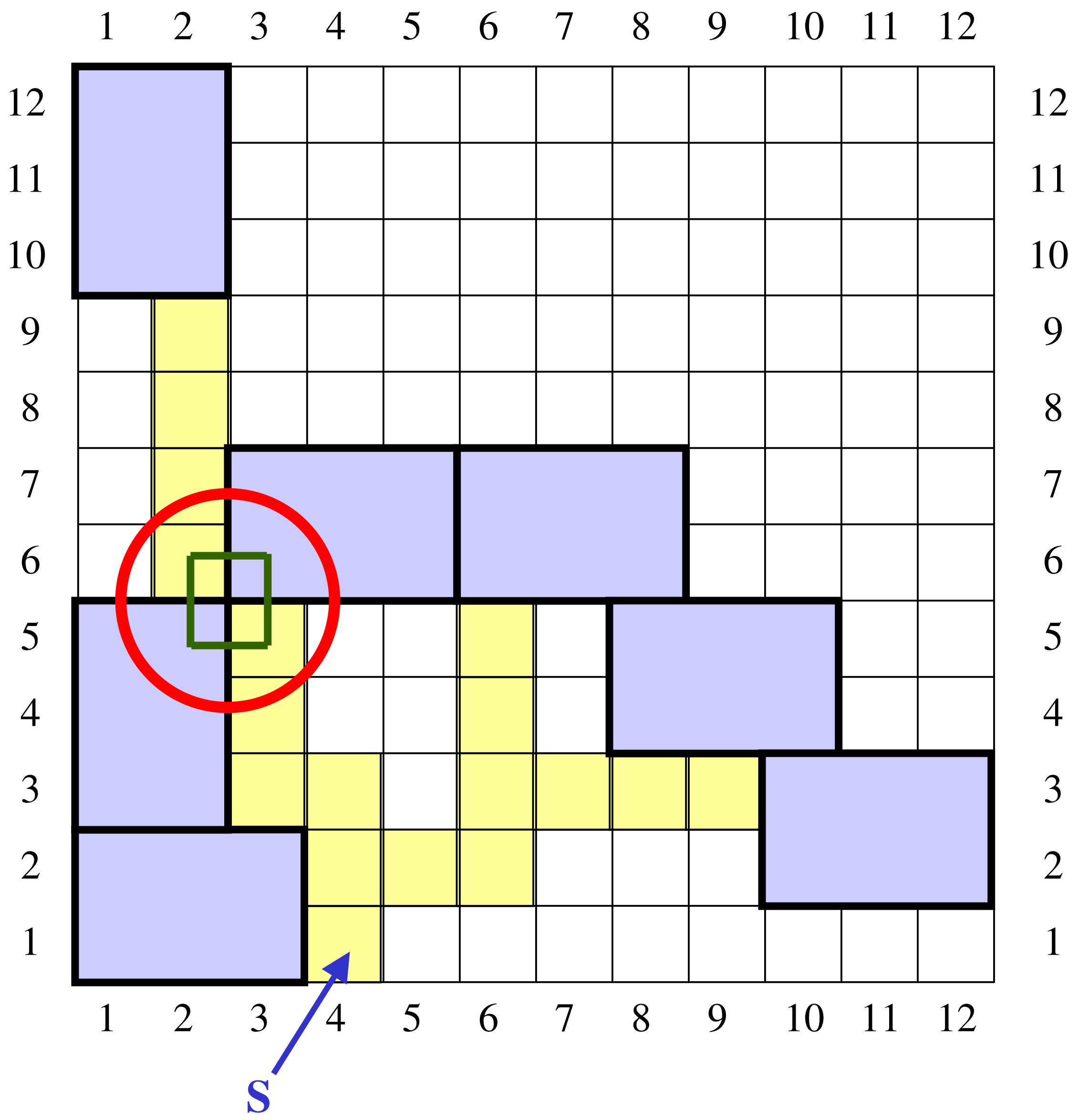
Diese Wege existieren vom Startfeld S zu den Häusern.

und wenn nun folgendes Haus eingefügt wird:



so würde bei ungerichtetem Graphen folgender Weg "erkannt":

Die Felder (3,5) und (2,5) sowie (3,5) und (3,6) sind laut Definition nun miteinander verbunden. Das gleiche gilt für die Felder (2,5) und (2,6) sowie (2,6) und (3,6).



Es gibt Wege für die unteren Häuser.

Folglich kann man durch die Häuser hindurch zu dem linken oberen Haus gelangen.

Fehler!

Wir müssen also den Kanten eine Richtung geben: Sie dürfen in Häuser hineinführen, aber nicht wieder heraus. Wir formulieren also neu (siehe oben):

Hierfür verwandeln wir die vorliegende Situation in einen ungerichteten Graphen: Die Felder des Baugrundstücks werden die Knoten des Graphens und zwei Knoten (x,y) und (x',y') werden genau dann durch eine ~~ungerichtete~~ gerichtete Kante von (x,y) nach (x',y') verbunden, wenn
- die Felder (x,y) und (x',y') rechteckig nebeneinander liegen und
- situation[x,y] ~~oder situation[x',y']~~ keine natürliche Zahl ist.

Da dieser Unterschied nur im Dijkstra-Algorithmus, der gerichtete und ungerichtete Graphen gleichermaßen bearbeiten kann, benötigt wird, spielt er im gerade betrachteten Zusammenhang keine Rolle und wir fahren fort, die rekursive Prozedur anzugeben.

Wir geben jetzt die Prozedur BtFp mit Ausnahme des Kürzeste-Wege-Verfahrens an. Zuvor ziehen wir die drei ersten Bedingungen als Boolesche Prozedur heraus. Hierbei testet die Boolesche Funktion ueberlappungsfrei(i), ob das i-te Feld des neu einzufügenden Hauses konfliktfrei mit der bisherigen Situation ist. Die hierauf aufbauende Boolesche Funktion konfliktfrei testet, ob zusätzlich die vierte Bedingung erfüllt ist; sie arbeitet nur mit globalen Variablen.

```
function ueberlappungsfrei (i: natural): Boolean;
begin
      ueberlappungsfrei :=
            (1 <= haus[i][1])  and
            (haus[i][1] <= n)  and
            (1 <= haus[i][2])  and
            (haus[i][2] <= n)  and
            ( (situation [haus[i]] = frei) or (situation [haus[i]] = weg) )
end;
```

Diese Funktion vollzieht exakt die ersten drei Bedingungen für das i-te Feld des neu einzufügenden Hauses "haus" nach.

```
function konfliktfrei: Boolean;
var i: natural; test123, test4: Boolean;
begin
    {m ist die Größe des Musters z;  haus := z + (f - z[1])}
    {Teste die ersten drei Bedingungen und weise das Ergebnis "test123" zu.}
    i:=1;
    while (i<m) and ueberlappungsfrei(i)  do  i := i + 1 od;
    test123 := ueberlappungsfrei(i);  test4 := false;
    if test123 then
    {Teste die vierte Bedingung und weise das Ergebnis "test4" zu.}
        "füge haus in die Situation ein und führe den Dijkstra-
            Algorithmus durch";
        test4 := "jedes Haus ist vom Startfeld aus erreichbar" fi;
    konfliktfrei := test123 and test4
end;
```

Nun präzisieren wir weitere Details von Algorithmus 6.3.

In der Prozedur müssen wir die Felder des Baugrundstücks durchlaufen. Wir hatten bereits festgelegt, dass dies "zeilenweise" in der Reihenfolge (1,1), (1,2), (1,3), ..., (1,n), (2,1), (2,2), ..., (2,n), (3,1), ..., (3,n), ..., (n,1), ..., (n,n) geschehen solle.

<u>for</u> <u>all</u> Feld f des Baugrundstücks <u>do</u> ....

schreiben wir daher mit den Komponenten kx und ky als

```
for kx:=1 to n do
  for ky := 1 to n do
     f := (kx,ky);  ....
```

Alle Muster sollen in der Menge ZulMust der "zulässigen Muster" abgelegt werden. ZulMust wird, wie bereits früher beschrieben, als ein array angelegt. Mit jedem Muster ist seine Größe (=Anzahl der Felder des Musters) anzugeben. Diese Werte werden im array "groesse" abgelegt:

<u>var</u> ZulMust: <u>array</u> [1..AnzMust] <u>of</u> Muster
groesse: <u>array</u> [1..AnzMust] <u>of</u> natural

groesse[i] gibt also die Anzahl der Felder, die zum i-ten Muster ZulMust[i] gehören, an.

<u>for</u> <u>all</u> z ∈ ZulMust <u>do</u> ....

schreiben wir mit der Laufvariablen nr als

<u>for</u> nr:=1 <u>to</u> AnzMust <u>do</u>
z := ZulMust[nr]; m := groesse[nr]; ....

Dann erhalten wir aus Algorithmus 6.3:

Algorithmus 6.4:

```
procedure BtFp;
var f, z: Muster; kx, ky, nr, m: natural;
begin
  for kx:=1 to n do
    for ky := 1 to n do
       f := (kx,ky);
       for nr:=1 to AnzMust do
         z := ZulMust[nr]; m := groesse[nr]; haus := z + (f - z[1]);
            if konfliktfrei then
                   setze Situation neu;  BtFp;
                   mache diese Setzung wieder rückgängig
            else evtl. lag eine Lösung vor, bewerte sie usw. fi
       od
  od od
end;
```

Nun fehlt im Wesentlichen nur noch der Dijkstra-Algorithmus für die Entscheidung, ob bei der gewählten Lage von "haus" alle bisher bereits platzierten Häuser noch vom Startfeld aus erreichbar sind. Dieser Algorithmus liefert zugleich die (Felder der) Wege, die zu den Häusern führen und in "situation" gespeichert werden. Dabei gehen die alten Informationen der "situation" verloren, die daher zuvor gerettet werden und nach Rückkehr aus der Rekursion wiederhergestellt werden müssen. Dies realisieren wir am einfachsten durch eine lokale Variable "altesituation", in der die bisherige Situation zu Beginn der Prozedur zwischengespeichert wird.

Hiermit ist die Formulierung als Programm abgeschlossen. Das vollständige Programm findet sich am Ende dieses Abschnitts.

5.6.6 Abschätzungen, Kritik

Wir haben gezeigt, wie sich das Füllproblem grundsätzlich lösen lässt, und wir haben einen Lösungsalgorithmus in Form der rekursiven Prozedur BtFp (Algorithmus 6.4 zuzüglich der Hilfsfunktionen und des Dijkstra-Algorithmus) angegeben. Bevor wir die letzten Details ergänzen, wollen wir abschätzen, wie aufwändig dieses Verfahren ist und welche anderen Vor- und Nachteile es besitzt.

Zunächst berechnen wir die maximale Rekursionstiefe R. Es sei Hmax die maximale Zahl an Häusern, die auf das Baugrundstück platziert werden können. Wir hatten in Abschnitt 5.1 bereits gesehen, dass (mit $G = n \cdot n$ und m' = maximale Höhe oder Breite eines Musters und m = minimale Größe eines Musters)

$$\text{Hmax} \leq (G - 3 \cdot (n - 2m')) / m \in O(n^2/m)$$

gilt. Der Abbruch der Rekursion geschieht im schlechtesten Fall nach Hmax+1 ineinander geschachtelten Aufrufen. Also gilt

$$R = \text{Hmax} + 1 \in O(n^2/m).$$

In jeder Rekursion werden AnzMust Muster auf genau $G = n^2$ Felder gesetzt, d.h., es werden maximal $AnzMust \cdot n^2$ Versuche (also: rekursive Aufrufe) durchgeführt. Wir müssen also im schlechtesten Fall mit

$$1 + AnzMust \cdot n^2 + (AnzMust \cdot n^2)^2 + (AnzMust \cdot n^2)^3 + \ldots$$

$$\ldots + (AnzMust \cdot n^2)^R \in O((AnzMust \cdot n^2)^{Hmax})$$

Aufrufen rechnen.

In jedem Versuch wird der Dijkstra-Algorithmus ausgeführt, der im planaren Fall proportional zu $v \cdot \log(v)$ ist, wobei v die Zahl der Knoten des Graphens ist. In unserem Fall ist $v = n^2$, das heißt, der Dijkstra-Algorithmus kostet jedes Mal
$O(n^2 \cdot \log(n^2)) = O(n^2 \cdot 2 \cdot \log(n)) = O(n^2 \cdot \log(n))$ Schritte.
Das "Retten" der aktuellen Situation und das sich an den Aufruf anschließende Zurücksetzen kostet höchstens $O(n^2)$ Schritte.

Dies sind "die schlimmsten Faktoren" in unserem Algorithmus. Somit ergibt sich insgesamt ein Laufzeitverhalten von

$$O(n^2 \cdot \log(n) \cdot (AnzMust \cdot n^2)^{Hmax}) = O(n^2 \cdot \log(n) \cdot (AnzMust \cdot n^2)^{n^2/m}).$$

In dieser Größenordnung liegt also die Anzahl der Schritte, die unser Backtrackingverfahren bis zum Ende benötigt.

Das sind indiskutabel große Laufzeiten!

Rechnen Sie nach, dass im Falle von n=10, m=5 und AnzMust=2 der Wert von $n^2 \cdot \log(n) \cdot (AnzMust \cdot n^2)^{n^2/m}$ größer als 10^{45} ist. Selbst wenn wir Computer hätten, die 10^{12} Operationen in der Sekunde durchführen könnten, so müssten wir immer noch mehr als 10^{25} Jahre auf das Ergebnis warten (das Alter des Universums wird auf deutlich weniger als 10^{11} Jahre geschätzt).

Auch für unsere im Beispiel gewählte Grundstücksgröße von 144 (n = 12) und Mustergröße m = 6 werden weder wir noch unsere überschaubare Nachkommenschaft das Ende des Algorithmus erleben.

Zur Diskussion dieser Abschätzungen: Können diese Extremwerte überhaupt eintreten oder handelt es sich um Grenzen, die in der Praxis nicht erreicht werden können?

Die Zahl der Rekursionsaufrufe ist keine unrealistische Schranke, sondern eine recht genaue Abschätzung. Auch der Aufwand für den Dijkstra-Algorithmus ist in der Praxis nicht zu verringern. Dagegen wird die Zahl der rekursiven Aufrufe mit wachsender Rekursionstiefe immer geringer, weil dann die meisten Positionen nicht konfliktfrei sind. Doch wird dies den Exponenten nicht allzu sehr verringern. Der befürchtete Aufwand wird also in der Realität eintreten, d.h., *der vorgeschlagene Algorithmus 6.4 ist für die Praxis untauglich* (außer für relativ kleine Probleme).

Konsequenz: Wir müssen einen "besseren" Algorithmus finden.

Hier bieten sich folgende Vorgehensweisen an:

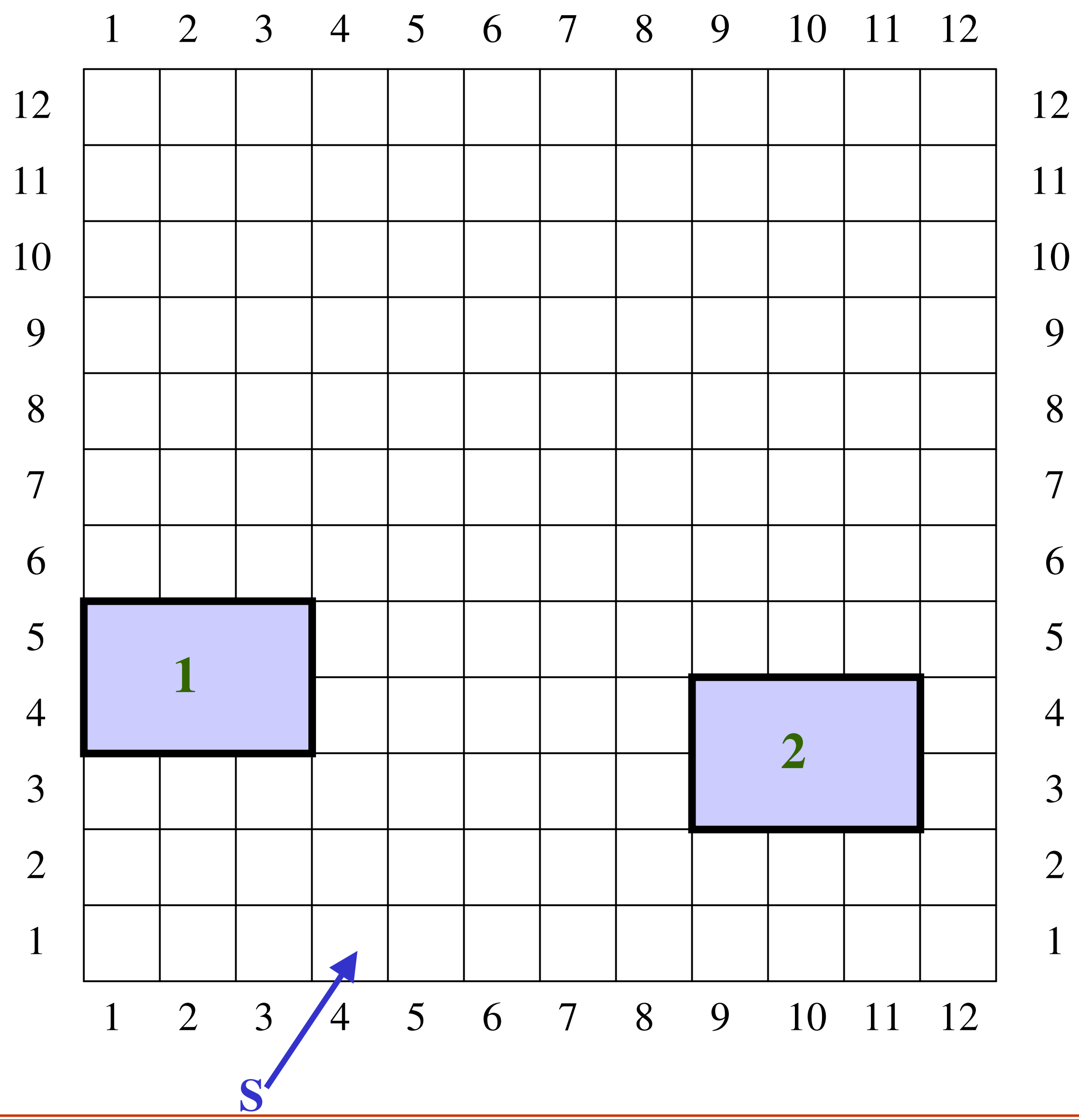
1. Wir überarbeiten unseren Algorithmus mit dem Ziel, möglichst viele Rekursionen, die nicht erfolgreich sein werden, zu vermeiden ("Branch-and-Bound"-Techniken).
2. Wir suchen nach anderen Verfahren. Doch hier werden wir vermutlich keinen Erfolg haben, da das Füllproblem, wie mehrfach erwähnt, NP-hart ist und daher nach heutiger Kenntnis jeder exakt arbeitende Algorithmus mehr als polynomielle (vermutlich sogar exponentielle) Laufzeit benötigt.
3. Wir verzichten darauf, die beste Lösung finden zu wollen, und verwenden Verfahren, die nur eine recht gute Lösung in kurzer Zeit aufspüren (Näherungsverfahren, Heuristiken).

Auf Näherungsverfahren gehen wir hier nicht ein.

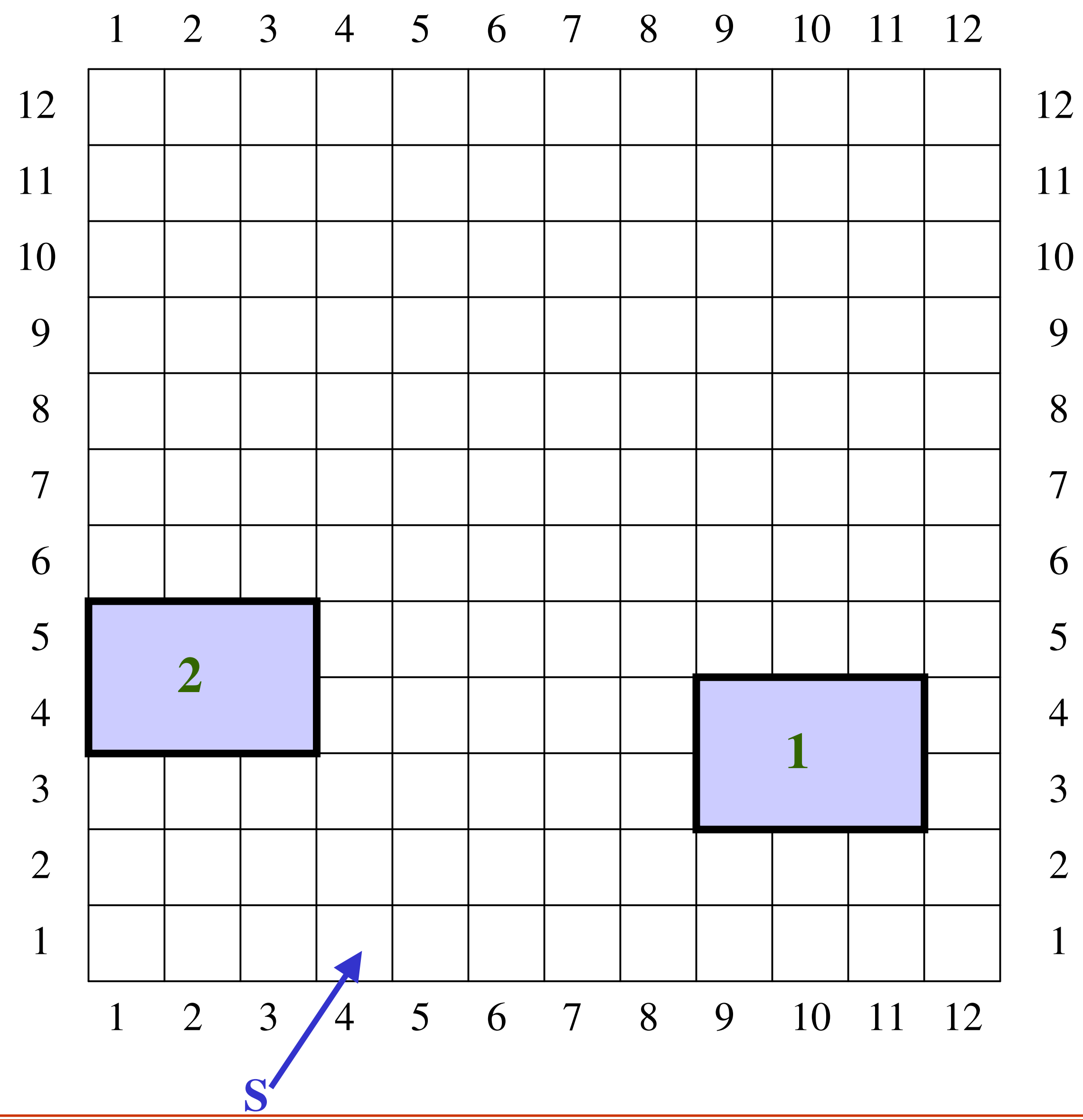
Im Folgenden werden Hinweise gegeben, wie man das Backtrackingverfahren für die Praxis (etwas) verbessern kann.

Dazu betrachten wir zwei Reihenfolgen beim Platzieren von Mustern. Das gleiche Muster soll beim ersten Mal zuerst im linken und danach im rechten Teil des Grundstücks und beim zweiten Mal in der umgekehrten Reihenfolge eingefügt werden.

Erst im linken, dann im rechten Teil:



Umgekehrt: Erst im rechten, dann im linken Teil:



Die rekursive Prozedur BtFp vollzieht *beide* Reihenfolgen nach. Dies ist nicht sinnvoll, da beide Fälle auf die gleiche Situation führen: Zwar unterscheiden sich die jeweiligen Felder dadurch, dass einmal mit "1" und einmal mit "2" belegt sind, aber da es sich um das gleiche Muster handelt, wird hier die gleiche Musterbelegung zweimal durchprobiert.

Diese Überlegung gilt auch für das dreimalige, viermalige ... Setzen eines oder auch mehrerer Muster. Wie viele sinnlose Wiederholung der immer gleichen Situation entstehen hierdurch?

Betrachten wir zunächst nur alle Platzierungen mit dem gleichen Muster: Die Prozedur BtFp prüft alle Permutationen, in denen die Muster auf das Grundstück gelegt werden können! Wenn man dies verhindern kann, würde sich die Laufzeit beträchtlich verringern. Und man kann es weitgehend verhindern, indem man das nächste Muster stets nur in einer kleinen Umgebung der bisher belegten Flächen platziert (siehe die rote Fläche auf Folie 148).

Wir wollen dies nicht ausformulieren, sondern uns nur überlegen, was man hierdurch an Geschwindigkeit gewinnen kann.

Hierzu müssen wir die Stellen betrachten, an denen das "haus" platziert wird.

Die Schleifen in der Prozedur BtFp

<u>for</u> kx:=1 <u>to</u> n <u>do</u>

<u>for</u> ky := 1 <u>to</u> n <u>do</u> f := (kx,ky); ...

bewirken, dass jedes Muster auf n^2 Positionen gesetzt wird. Betrachtet man aber *nur das Randgebiet der bisher gesetzten Häuser,* so stehen im Mittel höchstens noch n·m' Positionen zur Verfügung. Hierdurch würde sich die Zahl der rekursiven Aufrufe von $O((AnzMust \cdot n^2)^{Hmax})$ auf $O((AnzMust \cdot n \cdot m')^{Hmax})$ verringern.

Da m' (= die maximale Höhe bzw. Breite eines Musters) in der Regel sehr viel kleiner als n ist, lässt sich die Prozedur eventuell doch für praxis-relevante Aufgabenstellungen verwenden.

Wir rechnen dies für ein Beispiel durch.

Beispiel: Im Falle von n=10, m=5, m'=3 und AnzMust=2 würde der Wert von $n^2 \cdot \log(n) \cdot (AnzMust \cdot n \cdot m')^{n^2/m}$ kleiner als 10^{32} sein. Gegenüber der vorigen Berechnung wäre dies eine Verringerung ungefähr um den Faktor 10^{13}. Kleinere Probleme (vor allem solche mit nur einem Muster) könnten nun durchaus gelöst werden, auch wenn unser Beispielproblem mit n=12, m=6, m'=3 und AnzMust=2 vermutlich immer noch nicht effizient in Angriff genommen werden kann.

Was ist die Konsequenz aus dieser Untersuchung?

Statt der Schleifen

```
  for kx:=1 to n do
    for ky := 1 to n do   f := (kx,ky); ...
```

sollten zu Beginn der Prozedur BtFp die "sinnvollen" Felder f, auf die das Leitfeld jedes Musters gelegt wird, in der Umgebung der bereits bestehenden Häuser berechnet werden.

Dieses Vorgehen ist sicher vernünftig, aber die genaue Bestimmung der "Umgebung der bereits bestehenden Häuser" und der "sinnvollen Felder" ist außerordentlich schwierig. Denn man muss beweisen, dass hierbei die optimale Lösung auf keinen Fall verloren geht.

Wir brechen an dieser Stelle ab; denn es beginnt nun ein allgemeines Phänomen: Will man ein Problem möglichst gut lösen, dann muss man die spezifischen Eigenheiten des Problems in den Lösungsalgorithmus einfließen lassen! Man untersucht also keine breit einsetzbare Methoden mehr, sondern konzentriert sich auf problemspezifische Fragen. Solch eine Spezialisierung gehört aber nicht mehr zur Vermittlung des allgemeinen Backtracking-Prinzips, sondern sie wird entweder in Spezialveranstaltungen (Fachpraktikum, Studienarbeit usw.) oder erst nach Beendigung der Ausbildung gelehrt und gelernt.

Das soll Sie aber nicht hindern, den Algorithmus vollständig auszuformulieren, in eine Programmiersprache zu übertragen, auszutesten und an geeigneten Beispielen genau zu untersuchen.

Ende des Kapitels 5.