Zusammenfassung

Optimierungsalgorithmen
WiSe 2024/25

1 Introduction

1.1 (Euklidisches) Travelling Salesman Problem

- Input: endliche Menge von Punkten in einer Ebene
- Zulässiger Output: ein geschlossener Zyklus auf den Punkten
- Ziel: Länge des Zyklus (=Summe der Kantenlängen) minimieren

1.2 Steiner Tree Problem

- Input: Menge von Punkten K in der Ebene ($x \in K; x =$ Terminal)
- Gewünschter Output: Ein Netzwerk finden, welches alle Terminals verbindet und minimale Gesamtlänge hat
- Problem als Graph:
  - Input: Ein Graph $G = (V, E)$, ein Menge an Terminals $K \subseteq V$, Kantengewichte $w_e \geq 0, \forall e \in E$
  - Gewünschter Output: Eine verbunder zyklus-freier Subgraph $T = (V', E')$ von G mit $K \subseteq V'$ und minimales Gesamtgewicht $\Sigma_{e \in E'} w_e$

$\Rightarrow$ STP ist eine Abwandlung von MST (welches polynomial ist). STP ist jedoch NP-hart, weil es Terminals und Nicht-Terminals gibt.

1.3 Typen von algorithmischen Problemen

- Entscheidungsproblem: Gibt es eine Lösung?
- Konstruktionsproblem: falls eine Lösung existiert, erstelle eine.
- Enumerationsproblem: Gebe alle Lösungen an.
- Optimierungsproblem: erstelle eine (fast) optimale Lösung bzgl. eines Ziels.

1.3.1 Zutaten eines Optimierungsproblems

- zulässiger Input

- zulässiger Output
- Ziel

Formal definiert: Gegeben sei eine Menge I von möglichen Inputs. Für jedes Input $i \in I$ ist eine Menge F_i von zulässigen Lösungen definiert und eine Zielfunktion $\text{obj}_i : F_i \to \mathbb{R}$ und eine Richtung: Minimiern oder Maximieren.
Aufgabe: Festellen ob $F_i \neq$ und falls ja, finde $x \in F_i$ sodass gilt: $\text{obj}_i(x) = \begin{cases} \min\{\text{obj}_i(y) | y \in F_i\} \\ \max\{\text{obj}_i(y) | y \in F_i\} \end{cases}$

1.4 Matching

$\Rightarrow$ finde alle Kanten in einem Graphen, sodass keine dieser Kanten einen Knoten teilt.

Sei $G = (V, E)$ ein ungerichteter Graph. Ein zulässiger Output ist eine Menge $M \subseteq E$ sodass nicht zwei Kanten in M einen Knoten gemeinsam haben. $\leftrightarrow$ M ist ein *matching* von G.
Ziel: maximieren von $|M|$.

1.5 Zulässigkeit und Boundedness

- Eine Instanz $i \in I$ eines algorithmischen Problems heißt **zulässig**, wenn $F_i \neq$, sonst ist es unzulässig.
- Eine Instanz $i \in I$ eines Minimierungs Problems ist bounded, wenn die Zielfunktion bounded über F_i von unten ist, sonst ist es unbounded.
  $\to$ für Maximierungsprobleme von oben bounded.
  - Boundedness Instanz $i \in I \not\equiv$ boundedness von F_i in der Mengentheorie

1.6 Exakt vs Approximation vs Heuristisch

- ein Algorithmus ist **exakt**, wenn es beweist werden kann, dass es eine zulässige Lösung findet, falls eine existiert. (bei Optimierungsproblem, nicht nur zulässige Lösung sondern optimale Lösung)
- ein Algorithmus heißt **approximativ**, wenn es eine Lösung findet, bei der bewiesen kann, dass es nicht weit weg von einer zulässigen Lösung ist. (bei Optimierungsproblem, nicht nur zulässige sondern optimale Lösung)
- ein Algorithmus heißt **heuristisch**, wenn es versucht eine (fast) zulässige Lösung zu finden, aber keine Qualitätsgarantie bewiesen werden kann (bei Optimierungsproblem: versucht es eine (fast) optimale Lösung zu finden)

2 NP-completeness (skipped in Lecture)

3 Nachbarschaft-basierte Ansätze

3.1 Was ist eine Nachbarschaft?

Ein Mapping $N_i : F : i \rightarrow 2^{F_i}$ von der Menge der zulässigen Punkte, zu Ihrer Potenzmenge. Hieraus kann ein Nachbarschafts Graph aufgestellt werden, indem die Knoten zulässige Lösungen sind, und diese verbunden sind, wenn Sie benachbart sind. $\rightarrow$ wenn die Nachbarschaft symmetrisch ist, dann kann der Graph ungerichtet sein. $y \in N_i(x) \Leftrightarrow x \in N_i(y), \forall x, y \in V_i$

3.2 Globale vs. Lokale Optima

Sei $i \in I$ eine Instanz:

- dann ist ein globales Minimum für i eine zulässige Lösung $s^* \in F_i$ sodass $\text{obj}_i(s) \geq \text{obj}_i(s^*)$ für alle $s \in F_i$.
- dann ist ein lokales Minimum für i eine zulässige Lösung $s^* \in F_i$ sodass $\text{obj}_i(s) \geq \text{obj}_i(s^*)$ für alle $s \in F_i$ mit $(s^*, s) \in A_i$

Eine Nachbarschaft Relation ist exakt, wenn alle lokalen Optima, globale Optima sind.

3.3 Allgemeine Lokales Suchschema

$\rightarrow$ Wenn folgendes Schema terminiert, wird ein lokales Optimum geliefert

1. Starte mit einer beliebigen zulässigen Lösung $s^* \in F_i$
2. Solange ein $s \in F_i$ existiert, mit $(s^*, s) \in A_i$ und $\text{obj}_i(s) < \text{obj}_i(s^*)$:
   - Suche ein s aus.
   - Sei s das neue s^*
3. Liefere s^* als finale Lösung

$\Rightarrow$ Wenn F_i endlich ist, wird der Algorithmus terminieren. Wenn die Nachbarschaft exakt ist, wird ein globales Optimum gefunden. $\rightarrow$ Nachbarschaftsgraph muss idR. nicht explizit gespeichert werden. Die Existenz einer Kante im Graphen wird konstituiert durch kleinere Modifikationen einer Lösung, sodass eine neue Lösung entsteht. **Zwei permutationen $\sigma_1 \neq \sigma_2$ von $\{1, 2, ..., n\}$ sind benachbart wenn Sie identisch sind außer:**

- ein Single Swap (ein Element Paar ist gewechselt)

$$\exists i, j \in \{1, ..., n\}, i \neq j, \forall k \in \{1, ..., n\} \setminus \{i, j\} : \sigma_1[k] = \sigma_2[k]$$

- die Permutationen unterscheiden sich bei einem Rotational Shift

$$\exists i, j \in \{1, ..., n\}, i < j, (\forall k \in \{1, ..., i-1, j+1, ..., n\} : \sigma_1[k] = \sigma_2[k]) \wedge \\ \forall k \in \{i, ..., j-1\} : \sigma_1[k+1] = \sigma_2[k]) \wedge \\ (\sigma_1[i] = \sigma_2[j])$$

  $\Rightarrow$ asymmetrische Nachbarschaftsrelation

3.4 k-opt-Nachbarschaft für TSP

$\Rightarrow$ je nachdem wie viele Kanten getauscht werden, desto unterschiedlicher sind die Nachbarschaften. (2 Kanten = 2-opt, ...) $\rightarrow$ Größe vom k-opt ist $\Omega(n^k)$ für TSP auf n Punkten. (In Praxis nur 2-opt und 3-opt angewendet)

3.5 Disjunkte Conditions

- Kanten-Disjunkt: Keine Pfade teilen eine Kante
- Knoten-Disjunkt: Keine Pfade teilen einen Knoten außer evtl. Endknoten (intern Knoten-Disjunkt)
- Kapazitäten:
  - Zusätzlicher Input: Eine Kante hat einen nicht-negativen Kapazitätswert
  - Anzahl der Pfade die diese Kante verwenden dürfen den Kapazitätswert nicht überschreiten

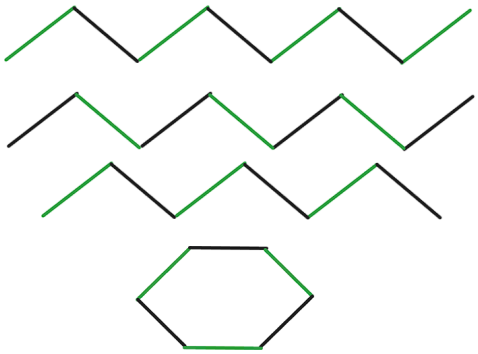
$\Rightarrow$ Nachbarschaften bei disjunkten Pfaden unangemessen, weil es wahrscheinlich ist, dass kein strikter Verbesserungsschritt machbar ist. Lokale Suche könnte nach wenigen Schritten fertig sein.

3.6 Exakte Nachbarschaftsprobleme

- Convex Programming
- Matching
  - eine Kante ist matched, wenn es im Matching ist
  - ein Knoten sit exposed/free, falls es in keiner gematchten Kante vorkommt
- Minimum Cost Flow

3.7 Alternierende Pfade

- ein Pfad p ist elementar wenn jeder KNoten nur einmal vorkommt
- ein elementarer Pfad ist alternierend, wenn genau jede zweite Kante in p Element von M ist.



$\Rightarrow$ Zwei Matchings M_1, M_2 in einem ungerichteten Graph $G = (V, E)$ sind benachbart wenn die symmetrische Differenz einen einzelnen alternierenden Pfad von M_1 und M_2 ist.

$$M_1 \triangle M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

3.8 Nachbarschaftsrelation für Matchings

Lemma: Diese Nachbarschaftsrelation ist exakt.

Seien 2 Matchings M_1, M_2, sodass gilt: $|M_2| > |M_1|$.
Zu zeigen: es existiert ein alternierender Pfad p für M_1, sodass M_1 verbessert wird:
$|(M_1 \triangle p)| > |M_1|$

Beweis:

- Durch die Strukturen alternierender Pfade, teilen M_1 und M_2 max. 1 Knoten.
  $\rightarrow$ jeder Knoten hat einen Grad von max. 2 in $M_1 \triangle M_2$
  $\rightarrow$ $M_1 \triangle M_2$ zerfällt in elementare Pfade die alternierend sind für M_1 und M_2.
- weil $|M_2| > |M_1|$, muss es ein p geben, welches mehrere Kanten von M_2 als M_1 besitzt.
  $\rightarrow$ $|(M_1 \triangle p)| > |M_1|$

3.9 Augmentierter Pfad

Ein alternierender Pfad, welcher bei nicht gematchten Knoten beginnen und aufhören.

3.10 Berges Theorem

$\Rightarrow$ ein Matching M ist genau dann ein maximales Matching, wenn es keinen augmentierenden Pfad bzgl. M gibt.

1. wenn es kein augmentierten Pfad gibt, ist M maximum.
   $\rightarrow$ Wahr, denn durch den augmentierten Pfad können alternierende Pfade vergrößert werden. Es gibt keinen augmentierten pfad, also kann das Matching nicht vergrößert werden.

3.11 Suche nach augmented Pfaden

$\rightarrow$ Breiten- und Tiefensuche können stecken bleiben aufgrund von Zyklen (Blossoms). Dieses Problem wird durch Blossom Algorithmus gelöst.
Blossom: ungerichteter Zyklus mit ungerader Anzahl von Knoten in einem Graphen
Blossom Shrinking: man trifft bei der Suche nach augmented Paths auf ein Blossom, dann schrumpft man diesen zu einem Knoten und sucht weiter. Im Nachhinein wird das Blossom wieder aufgebaut.

1. Finde Blossom
2. erstelle Pseudonode
3. finde augmented Pfad
4. verbessere Matching
5. Lift Pfad

3.12 Min-Cost-Flow ToDo

4 Heuristische Suche

ein lokales Suchschema garantiert nur eine Terminierung bei einem lokalen Optimum

4.1 Modifikationen

- Schritte von einer zulässigen Lösung zu einer weiteren, die eventuell Schlechter ist.
- Vorwärts Schritte durchführen und das beste Ergebnis aussuchen, von dem die Iteration weiter geführt wird
- wiederholen von lokalen Suchen für verschiedene Startlösungen
- Mehrere Lokale Suchen Simultan laufen lassen

4.2 Simulated Annealing

1. Starte mit beliebiger zulässigen Lösung $s^* \in F_i$
2. Solange eine bestimmte Terminierungskonidition nicht erfüllt wurde
   - Wähle zufällige zulässige Lösung $s \in F(i)$ sodass $(s^*, s) \in A_i$
   - Wenn $\text{obj}(s) < \text{obj}(s)$, sei s neues s^*
   - Andererseits
     - Mache eine Probabilistische Ja oder Nein Entscheidung
     - Wenn Ja, dann s ist das neue s^*
     - sonst lass s^* so wie es ist
3. Liefere soweit beste Lösung als finale Lösung

4.3 Probabilistische Ja/Nein Entscheidung

$\Rightarrow$ Biased Münzwurf. Verschiedene Wahrscheinlichkeiten für Kopf oder Zahl.
Random number generator: Ein deterministischer Zahlen Generator, der eine nicht-deterministische Wahl von Nummern simuliert.
Im Simulated-Annealing, wird die Ja-Wahrscheinlichkeit durch die Temperator $T > 0$ bestimmt.
Für $\text{obj}(s) \geq \text{obj}(s^*)$, is die Wahrscheinlichkeit für Ja

$$0 \leq exp(\frac{\text{obj}(s^*) - \text{obj}(s)}{T}) \leq 1$$

4.4 Cooling Schedule

- Wie wird T definiert?
- Wie wird die Terminierungskondition definiert?

Cooling Schedule:

- eine endliche Sequenz von Temperaturwerten wird definiert: $T_1 > T_2 > ... > T_k > 0$
- Für $i \in \{1, ..., k\}$ ist eine positiv ganzzahlige Folge der Länge n_i zusätzlich definiert
- **Anwendung** des cooling schedule:
  Für $i \in \{1, ..., k\}$, werden genau n_i Iterationen der While Schleife (siehe 4.2) mit T_i durchgeführt.

4.5 Terminierungs Kriterien

Um zu ermöglichen, die Prozedur frühzeitig abzubrechen, falls der Cooling Schedule zu Zeitaufwendig gewählt wurde, werden weitere Terminierungskriterien angewendet.
$\rightarrow$ Üblicherweise leidet dadurch die Ergebnisqualität

4.6 Konvergenz von Simmulated Annealing ToDo

4.7 Feature-basierte Lokale Suche

$\Rightarrow$ für eine Instanz gibt es eine endliche Grundmenge an Features. Die zulässigen Lösungen zu dieser Instanz sind gewisse Untermengen der Grundmenge.

- Features $\equiv$ Dimensionen
- Beispiele von Features
  - TSP: Features sind die Paare aneinander folgenden Knoten
  - Matching: Die Kanten des Inputgraphen sind Features.
  - Set covering:
    - Input: eine endliche Grundmenge F udn eine Collection S von Untermengen von F
    - Zulässiger Output: Eine Untermenge $S' \subseteq S$ sodass F von S' gedeckt ist

      $$\forall x \in F \, \exists s \in S' : x \in s$$

    - Ziel: $|S'|$ minimieren
    - Features: Elemente von S
  - Max-cut
    - Input: ein ungerichteter Graph $G = (V, E)$ und eine Kantengewichtung $c : E \to \mathbb{R}$
    - Zulässiger Output: beliebige Untermenge $W \subseteq V$
    - Ziel: Knoten zwischen zwei Gruppen aufzuteilen, sodass möglichst viele (oder stark gewichtete) Kanten zwischen W und V verlaufen.
    - Features: die Knoten
  - Coloring
    - Input: ungerichteter Graph $G = (V, E)$
    - Output: eine Zuweisung $C : V \to \mathbb{N}$ einer positiven Ganzzahl für jeden Knoten
    - Zulässige Lösung: Wenn $C[v] \neq C[w] \, \forall \{v, w\} \in E$
    - Ziel: Minimieren der größten zugewiesenen Zahl
    - Features: Paare (v, n) sodass $v \in V$ und $n \in \mathbb{N}$

4.8 Wahl des Feature Sets

$\Rightarrow$ Die Features der Probleminstanzen sind von der gewählten Problemformulierung abhängig
$\rightarrow$ typisch wird in Optimierungsproblemen jedem Feature eine Feature-Cost zugewiesen und die Kosten einer Lösung ist die Summe der Feature-Costs die die Lösung aufbauen

4.9 Sind alle Nachbarschaftsrelationen Feature-basiert?

$\rightarrow$ Nein. Zum Beispiel ist dies nicht wahr für alle algorithmischen Probleme, in der die zulässigen Lösungen unendlich sind.

4.10 Feature-Based Lokale Suche

Idee

- gewisse Features werden bestraft (Guided Local search)
- das Ändern von gewissen Features wird verboten (Taboo Search)

⇒ so ermöglicht man ein lokales Optimum zu verlassen

⇒ bei Optimierungsproblemen, bei denen die Kosten einer Lösung die Summe der Kosten der ausgewählten Merkmale sind, können die Kosten der Lösung L so geschrieben werden: $C[L] = \sum_{x \in F(S)} C[x]$

4.10.1 Guided local search

Wie das allgemeine Local-Search Schema, unterscheidet sich jedoch wenn es auf ein lokales Optimum trifft:

- Algorithmus betrachtet alle Features, die das lokale Optimum bilden
- Für jedes Feature wird ein "Nutzen der Bestrafung" bestimmt
- ein oder mehrere Features mit den höchsten Bestrafungsnutzen, werden bestraft
- die Bestrafung ist so enorm, sodass die Lösung nichtmehr als lokales Optimum betrachtet wird
- Lokale Suche wird normal weitergeführt

Penalized/Bestraft: Die Kosten des Features werden um den Penaltywert erhöht.
Utility of Penalization / Nutzen der Bestrafung: Einschätzung, wie erfolgsversprechend es wäre, dieses Feature zu bestrafen

- wenn die Originalkosten eines Features hoch sind, könnte es nützlich sein, mit einem Penalty das Feature direkt auszuschließen
- aber wenn ein Feature oft bestraft wurde, trotzdem wieder in der aktuellen Lösung ist, dann ergibt es evtl. nicht so viel Sinn das Feature zu bestrafen

4.10.2 Konfliktlösung zwischen Penalties

Für jedes Feature $x \in F_i$, wird die Anzahl getracked, wie oft dieses Feature bisher bestraft wurde. Dies wird in p_x gelagert. Wenn es schon j-mal bestraft wurde, wird ein Bestrafungswert definiert

λ_{xj}. Dann sind die Kosten einer zulässigen Lösung zu i (inkl. aller Bestrafungen) folgendes:

$$\tilde{C}[S] := \sum_{x \in F(S)} (C[X] + \sum_{j=1}^{p_x} \lambda_{xy})$$

Der Bestrafungsnutzen wird dann wie folgt definiert: $\frac{\tilde{C}[x]}{1+p_x}$

4.10.3 Taboo search

- geht immer zum Nachbarn mit geringeren Kosten, auch wenn aktuelle Lösung ein lokales Optimum ist
- Um das Algorithmus zu terminieren, muss ein externes Abbruchkriterium eingefügt werden (zB nach bestimmter Anzahl an Schritten)
- Problem hierbei ist es, dass der Algorithmus höchstwahrscheinlich ein lokales Optimum betritt, welches es zuvor verlassen hatte, wodurch eine Endlosschleife verursacht werden könnte
- um dies zu verhindern, werden die letzten Änderungen von Features für einige Schritte gespeichert. Die Änderung eines Features aus dem Move einer zulässigen Lösung s_k zu einer benachbarten Lösung s_{k+1} wird so dargestellt:

$$x \in F(s_k) \setminus F(s_{k+1} \text{ or } x \in F(s_{k+1}) \setminus F(s_k)$$

- Solange die Änderung "im Kopf gehalten wird", muss die Suche die Änderung nicht reversen
- Typisch (nicht exkl.): Eine Schrittzahl ist Global definiert, und jede Änderung wird von der Taboo Liste nach der Schrittzahl entfernt

4.10.4 Aspiration

Praktische Anwendungen von TabooSuche haben gezeigt, dass es ab und zu gut ist, alle Taboos abzuwerfen, um die Suche mit einer leeren Tabooliste weiterzuführen.
Eine Kondition das löschen der Tabooliste erzwingt, ist ein Aspirations Kriterium.

4.11 Allgemeines Konzept zu mehrere Vorwärtsschritte machen und die beste Lösung auwählen

- In jeder Iteration der While-Schleife wird ein gewisser Pfad aufgebaut im Nachbarschaftsgraphen, der mit der aktuellen Lösung s^* beginnt.
- Die andere beste Lösung auf dem Pfad wird als neue aktuelle Lösung ausgesucht.
- Falls die neue Lösung schlechter ist als die alte Lösung, dann wird die neue Lösung trotzdem angenommen, der Algorithmus terminiert oder der Algorithmus versucht einen neuen Pfad zu finden.

4.11.1 Variable Depth Search

Die Tiefe der Suche wird dynamisch angepasst. Wenn vielversprechende Verbesserungen gefunden werden, dann wird die Tiefe erweitert. Der Gain dient als Entscheidungskriterium, ob die Tiefe weiter erhöht wird oder nicht. Wenn der Gain unter einen bestimmten Werte fällt, dann wird abgebrochen.

4.11.2 Details der Standardprozedur

- Tabooliste wird entstand gehalten während Pfad konstruiert wird, anfangs war die Liste leer.
- Während ein Pfad aufgebaut wird, werden Taboos nie entfernt. Ein Pfad dessen Einfügung den Taboos verletzt, ist temporär unzulässig.
- bei jedem Schritt, wird der beste zulässige Arc genommen. Und der Pfadbau terminiert, wenn ein Knoten erreicht wird, dessen Arcs unzulässig sind.

4.11.3 Iterated/Chained Lokale Suche

- Starte mit beliebigen zulässigen Lösung $s^* \in F_i$
- Während das Terminierungskriterium nicht erfüllt wird, tue:
  - $s = \text{Pertubarbation}(s^*)$;
  - $s' = \text{LocalSearch}(s)$;
  - $s^* = \text{AcceptanceCriterion}(s^*, s')$
- Liefere s^* als die finale Lösung.

⇒ Perturbation: modifiziert die aktuelle Lösung und Liefert zulässige nächste Lösung. Nächste Lösung sollte nicht zu klein oder zu groß sein.

⇒ LocalSearch: kann ein belibiger Algorithmus sein, welche eine zulässige Lösung als Input bekommt und eine weitere als Output ausgibt (Kann eine Black Box sein

⇒ AcceptanceCriterion: entscheidet, ob wir $s^{'}$ als neue Lösung akzeptieren oder bei der bisherigen Lösung s^* bleiben

4.12 Multi-Start Local Search

Problem:

- Wahrscheinlichkeit hoch, dass die Suche einen kleinen Unterraum des Lösungsraum nie verlässt
- die sehr guten zulässigen Lösungen könnten wo anders im Lösungsraum sein

Einfachste vorstellbare Lösung:

- Generiere Menge von zulässigen Lösungen (zB random)
- Starte lokale Suche von jedem Element der Menge (oder Simulated Annealing, Taboo Search, ...)
- Liefere beste zulässige Lösung die durch eine der Suchen erzeugt wird

4.13 Populations-basierte Strategien

⇒ Multi-Start ist einfachste Populations basierte Strategie.

4.13.1 Survival of the Fittest

Generiere Menge von zulässigen Lösungen, und starte Suche von jedem Einzelnen Element.
Alle Suchen werden in Runden durchgeführt, simultan in pseudo-parallelität. Die besseren Suchen überleben und besten erzeugen "Nachkömmlinge).
In einer Runde wird ein Schritt um Graphen durchgeführt, und die aktuellen Suchergebnisse werden verglichen. Die Suchen mit schlechten Ergebnissen werden verworfen. Die besten Lösungen werden verzweigt in weitere Suchen.

4.13.2 Evolutionäre Algorithmen

Fitte Mitglieder der Population sind in der Lage sich durch Duplikation zu vermehren. Je fitter ein Mitglied ist, desto höher ist die Vervielfältigungswahrscheinlichkeit. Die Nachkömmlinge werden zufällig von der Nachbarschaft der Eltern gewählt. Populationsmitglieder werden zufällig von den Umständen verworfen. Je fitter ein Mitglied ist, desto unwahrscheinlich das Verwerfen.
Eine Runde:

- bestimmte Anzahl der Population wird zufällig ausgewählt mit einer Wahrscheinlichkeit, die monoton steigt.
- Die gewählten Mitglieder werden verworfen.
- weitere Mitglieder werden zufällig ausgewählt, mit einer monoton sinkenden Wahrscheinlichkeit. Diese Mitglieder erzeugen Nachfolger.

4.14 Genetic Algorithms

⇒ ähnlich wie Evolutionäre Strategien.

- Eine neue Generation wird von den ausgewählten Mitgliedern generiert im sinne der (asexuellen) Mutation.
- In genetischen Algorithmen wird jedes Mitglied der Kindgeneration durch zwei Mitgliedern der Elterngeneration erzeugt, durch (sexuelle) Rekombination.
- Die Suche findet nicht auf den zulässigen Lösungen statt, sondern deren Repräsentationen (Genen).
- Für alle Isntanzen sind die Gene Strings.

4.14.1 Rekombination von Genen

- Zwei Nachkömmlinge werden simultan durch ein Elternpaar erzeugt (deterministisch or randomized)
- Die Rollen von den Eltern werden in der Produktion von Nachkömmlingen getauscht.
  $\Rightarrow$ Recombination = Crossover

4.14.2 Qualität der Crossover strategie

- Fitness der Nachkömmlinge, sollte denen der beiden Eltern entsprechen
- wenn ein Feature zB in beiden Elternteilen vorkommt, dann sollte es auch im Kind sein
- wenn ein Feature in keinen Elternteilen vorkommt, dann sollte es auch nicht im Kind sein, weil es Spiegelsymmetrisch ist

4.14.3 Crossover Strategie

- One-point crossover: es wird ein Punkt ausgewählt entlang der Chromosomen. Die Eltern-Individuen tauschen den Teil hinter dem Punkt. Also erhält ein Kind die Chromosomen eines Elternteils bis zum Punkt, dann den hinteren Teil des anderen Elternteils and vice versa.
- Two-point crossover: zwei Punkte werden entlang der Chromosomen ausgewählt. Eltern tauschen den Abschnitt zwischen den beiden Punkten.
- Uniform crossover: Für jedes Gen wird zufällig entscheiden ob es vom ersten Elternteil kommt oder vom zweiten. Vererbungsentscheidung erfolgt anhan Uniformverteilung.

$\Rightarrow$ Lösungen könne jedoch unzulässig sein. Dies kann verhindert werden, indem alle Nebenbedingungen verfallen werden (nicht so nice), oder es bestraft wird, wenn Nebenbedingungen nicht eingehalten werden. Alternativ kann man die Repräsentation anders definieren, sodass so viele Bedingungen wie möglich erfüllt werden.

4.14.4 Resource-Constrained Scheduling

Input: eine endliche Menge an Jobs $J_1, ..., J_n$; eine Dauer d_i für jeden Job und eine Paarauswahl (J_i, J_j) sodass der Graph mit Menge $\{J_1, ..., J_n\}$ und die Ausgewählten Paare als direkte Kanten azyklisch sind. Zusätzlich irgendwine Resource Constraint.
Output: Ein Assignment für jeden Job zu einer Startzeit.
Constraint: $t_i + d_i \leq t_j$ für jeden $(J_i, J_j) \in S$ (zweiter Job startet erst wenn erster Job fertig [precendence Constraint]), und die Ressourcenconstraints (Gibt an wie viele Jobs gleichzeitig durchgeführt werden können).
Ziel: minimiere $\max_i\{t_i + d_i\}$

$\Rightarrow$ Diese Repräsentation mit Blick auf Gene ist recht schlecht, weil Rekombinationen zu unzulässigen Lösungen führen. Wenn man die Resourceconstraint ignoriert und das Verstoßen der Constraint bestraft, könnte zulässigkeit gewährleistet werden bei der Erzeugung weiterer Lösungen. Man definiert dann Slacks, welches die minimale Differenz der Startzeit des Jobs und dem auf das er Warten musste ist, und die Gene werden zB durch Slack-Values repräsentiert. Wenn der Slack nicht negativ ist, wird eine zulässige Lösung erzeugt.
$\Rightarrow$ Das ist eine bestimmte Crossover-und-Repair Strategie die für TSP angepasst ist (Partially Mapped Crossover)

4.14.5 PMX

Es nutzt two-point Crossover. Was jedoch repariert werden muss, ist dass im Substring vom Two-Point Crossover, oder eher in den Offsprings gleiche Städte vorkommen können.
Repairloop: Während zwei Städte doppelt vorkommen, wird folgendes angewendet:

- Sei $i \in \{l, ..., r\}$ und $j \in \{1, ..., l-1, r+1, ..., n\}$ sodass $O_1[i] = O_1[j]$
- Setze $O_1[j] := P_1[i]$

In worten: Wir suchen im Innenstück nach der Stadt und im Außenstück nach der selben Stadt. Dann tauschen wir im Außenstück die Stadt mit einer Stadt aus dem Elternteil aus (der an selber Indestelle ist). Wenn dadurch neue Doppelungen entstehen wird der Loop fortgeführt. Wenn man die Reparatur als Graphen betrachtet, terminiert der Loop immer, außer man geht entlang eines Graphen.

$\Rightarrow$ PMX hat eine allg. Crossover Strategie, aber die Reparatur ist spezifisch für Probleme in denen zulässige Lösungen Permutationen sind.

4.14.6 Order Crossover (OX)

Bei Problemen verwendet wo die Chromosomreihenfolge wichtig ist (bsp TSP). Es wird dafür gesorgt, dass die Reihenfolge der Eltern beibehalten wird, ohne dass Duplikate oder Konflikte

auftreten in Permutationen.
Schritte

- Zufällige Auswahl des Bereichs
- Segment kopieren in das erste Kind
- restliche Gene auffüllen, mit der (relativen) Reihenfolge des zweiten Elternteils
- Zweites Kind wird entsprechend erzeugt

4.15 Mutation in Genetischen Algorithmen

Jede Runde eines genetischen Algorithmus' hat folgende Schritte:

- eine gewisse Anzahl an Paaren von Eltern wird für Reproduktion ausgewählt
- Die Paare werden Kombiniert und formen Nachfolgen.
- Mit geringer Wahrscheinlichkeit, wird ein Nachfolge mutiert wie in Evolutionsstrategien

4.16 Kulturelle Algorithmen

⇒ In genetischen Algorithmen gibt es nur Soziale Aktivität von Eltern. Wenn individuelle Mitglieder einer Population aufeinander reagieren (positiv oder negativ) und dass bessere Mitglieder mehr positive Reaktionen anderer Mitglieder bekommen. Diese Algorithmen werden kulturelle Algorithmen genannt.

4.16.1 Ameisenkolonie Optimierung

Eine Ameise findet kürzeste Pfade durch Pheromone, die von vorherigen Ameisen verteilt werden. Die Ameise folgt dem intensivsten Pfad. Je kürzer ein Pfad, desto Wahrscheinlicher ist es, dass eine zufällig rumlaufende Ameise diesen Weg genommen hat. Der Pfad gibt nur die Wahrscheinlichkeit dass eine Ameise dem Pfad folgt (wichtig um lokalen Optima zu vermeiden)

- Positives Feedback: Jede Ameise simuliert andere Ameisen umdenen zu folgen
- Negatives Feedback: Pheromone in einem Pfad verringern permanent. Die nächste Ameise refreshed den Pfad, und wenn ein Pfad nicht oft verwendet wird, sinkt die Wahrscheinlichkeit mehr.

Umformulierung in einen Algorithmus Jede Instanz soll einem gerichteten Graphen entsprechen und jede Lösung einer Instanz ist ein Pfad im Graph. Nebenbedingung des Problems sind Nebenbedingung des Pfades und die Kostenfunktion könnte als der Länge des korrespondierenden Pfades

beschrieben werden.
Idee: Ameisen bewegen sich im Graph, Pheromon-Mechanismus wird simuliert, und hoffentlich werden die Ameisen eher kürzeren Wegen folgen.

4.16.2 komplexere Nebenbedingungen

Alle allg. Lokalen Suchschemen sind (irgendwie) abhängig auf eine passende Nachbarschaftsstruktur. Fundamentale Lokales Suchschema bneötigt eine einfache enumerable Nachbarschaftstruktur, und Simmulated annealing benötigt eine Nachbarschaftsstruktur in der zufällige Auswahl einfach ist.

4.17 Problemzusammenfassung

Die Nachbarschaftsrelation kann sehr dünn sein (vielleicht sogar getrennt). Durch lose Nachbarschaftsbeziehungen ist es wahrscheinlich dass die Suche in kleinen Submengen im Suchraum sind (um der Startlösung herum). Wahrscheinlichkeit ist hoch, dass die Suche schnell in einem lokalen Optimum stecken bleibt.
Bei Algorithmen ohne nummerierten Nachbarschaften aber mit zufällige gewählten Nachbarn, könnte es etwas dauern, bis man eine zulässige Lösung findet.

Typische Ansätze:

- Relaxieren von Nebenbedingungen (Bedingungen droppen und Bestrafungen einführen)
  - Probleme: Man findet zwar gute Lösungen, aber es ist sehr langsam. Es könnte lange dauern, bis die erste Zulässige Lösung gefunden wird. Wenn zulässige Lösungen gefunden werden, dann sind das in der Regel sehr gute.
- Variable Gewichte in Bestrafungen (Bestrafung wird durch positiven Penalty Faktor multipliziert; Faktor wird während der Laufzeit größer, also unwahrscheinlicher das Fehler passieren)
- Wahrscheinlichkeit ist hoch, dass die Zielfunktion eher "Smooth" auf der Menge der zulässigen und unzulässigen Lösungen ist. (gute zulässige Lösungen werden eher dort gefunden wo die Zielfunktion gut auf zulässigen und unzulässigen Lösungen ist)

4.18 Algorithmische Zielsetzung

1. Optimierung eine möglichst Gute Lösung zu finden (Optimierung ¿ Zulässigkeit)
2. Möglichst nah an die Zulässigkeit zu kommen (Zulässigkeit ¿ Optimierung)

5 Entscheidungsbasierte Ansätze

⇒ Entscheidungen die getroffen werden, verkleinern den Lösungsraum. Die Suche erstellt aus dem Suchraum eine Baumstruktur (Search Tree). Im Suchbaum korrespondiert die Wurzel zum gesamten Lösungsraum. Jeder Knoten korrespondiert zu einer gewissen Untermenge aller Lösungen.

5.1 Branching eines Suchbaums

Sei S der Gesamte Lösungsraum der zu Knoten P gehört.

- Wir branchen auf einer binären Variable $x_i \in \{0, 1\}$: P hat genau zwei Kinder Q_0 und Q_1. Q_0 sind aller Kanten die nicht im Lösungsraum sind, und Q_1 sind alle Kanten die im Suchraum sind.
- Mit Kontinuierlichen Variablen kann man verschiedene Mengen definieren, zum Beispiel $< 0, \geq 5, ...$

5.2 Entscheidungsbäume

⇒ eine im feature-based Fall eine natürliche Definition von Suchbäumen. Hierbei wird entschieden ob ein Feature mitbetrachtet wird oder nicht. Auf Baum Level i entscheiden wir ob ein Feature ausgewählt wird oder nicht. Dadurch wird ein voller Binärer Baum erzeugt, in dem alle Blätter auf dem selben Level sind und jeder Knoten genau zwei Kinder hat.

Man kann dann den Lösungsraum so partitionieren in die Partiotion wo ein Feature drinnen ist, und wo es nicht drinnen ist, etc. Die Partitionen entsprechen einem Lösungsraum, die zu all den gewählten Features passt. Ein Knoten korrespondiert zu der Menge aller Lösungen die die bisher gewählten Features enthalten und nicht die Features enthalten, die bisher abgelehnt wurden.

5.2.1 Entscheidungsbäume entdecken

Typischerweise ist die Lösungsanzahl mind. exponentiell. Ein Entscheidungsbaum für eine Instanz ist also zu groß. Hierbei gibt es zwei Strategien: Die Entdeckung ist nur potentiall exhaustive und heuristisch eingeschränkt zu einem bestimmten Teil des Baums.

Potentially Exhaustive heißt, man versucht es zu beweisen, dass es nicht optimal ist, von einem Knoten weiter in den Baum abzusteigen, sodass dieser Knoten ignoriert wird und man Berechnungszeit spart (Pruning).

5.3 Allgemeine Algorithmusschema

1. Erstelle Wurzelknoten r der das originale Problem repräsentiert. Markiere Knoten als "Nicht-Besucht".
2. Solange ein nicht-besuchter Knoten existiert, tue folgendes:
   (a) Wähle einen nicht-besuchten Knoten n aus zur untersuchung. Markiere Knoten als besucht.
   (b) entweder:
   - Löse das Problem welches mit n assoziiert ist (finde heraus ob es unzulässig ist oder suche optimale Lösung)
   - oder entscheide den Subbaum mit Wurzel n zu abzuschneiden (Prune)
   - oder Branch, also Problem in Sub-Probleme und füge korrespondierende nicht-besuchte Knoten zum Baum hinzu

5.3.1 Priority Queue

Datenstruktur, welche es erlaubt folgende Operationen auf eine Kollektion H (wobei jedes Item ein Key hat)

- create-pq(H): Erstelle leere Priority Queue H
- insert(H, x): Füge Element x in H ein
- find-min(H, x): finde und gebe das Objekt x zurück, mit minimum Key in H
- delete-min(H, x): lösche Objekt x mit minimum Key von H
- decrease-key(H, x, y): verringere Schlüssel von Objekt x in H zu y

5.3.2 Vor- und Nachteile von Pure Strategies

- DFS:
  - Kann die erste zulässige Lösung recht schnell finden
  - Benötigt nur wenig Speicher
- Best-First:
  - Könnte helfen bessere zulässige Lösungen schnell zu finden
  - Größeren Overhead pro Iteration
- Breadth-FS:
  - Braucht sehr viel Speicher
  - Kann nützlich sein in kombinierten Strategien

5.3.3 Complete Enumeration

Die Suchräume sind idR zu groß um den ganzen Baum zu untersuchen. Dies ist jedoch Zwangsweise zu tun in Enumerationsproblemen.
Enumerationsprobleme: Wir möchten alle zulässigen Lösungen zurückgeben (nur sinnvoll bei endlichen Lösungsmengen)
Zählprobleme: Wir möchten die Anzahl aller zulässigen Lösungen zurückgeben
⇒ in diesen Fällen ist das beste was erwartet werden kann polynomial in der Größe der Inputs und Outputs. Bei Enumerationsproblemen ist Tiefensuche way to go.

5.3.4 Brute Force

⇒ wenn ein Problem gelöst wird, in dem alle Möglichkeiten im Entscheidungsbaum durchsucht werden

5.4 Die Kraft von Preprocessing

⇒ Transformiere eine Instanz I_1 eines Problems der Klasse P in eine Instanz I_2 der gleichen Problemklasse, sodass

- die Größe von I_2 kleiner ist als I_1
- wenn wir für I_2 eine optimale Lösung kennen, dann können wir einfach eine optimale Lösung für I_1 berechnen

⇒ Beispiel: Cardinality Matching
Sei $G_1 = (V, E)$ ein ungerichteter Graph in dem die maximale Kardinalitätsmatching gesucht wird.
Transformationsregel: Sei v ein Knoten mit Grad 1 in G_1, und (v, w) eine incident-edge. Lösche Knoten v und w (und alle incident-edges) von G_1 um Graph G_2 zu bekommen.
Wenn M_2 das maximale Kardinalitätsmatching in G_2 ist, dann gilt: $M_1 = M_2 \cup \{(v, w)\}$ ist ein maximales Kardinalitätsmatching in G_1.
Weiteres Beispiel: Hitting Set
Input: nicht-leere-endliche Menge F und Kollektion S von Untermengen von F.
Zulässiger Output: eine Untermenge $F' \subseteq F$ von F sodass $s \bigcap F' \neq$ für jedes $s \in S$.
Ziel: minimiere $|F'|$

5.4.1 Hypergraphen und Hittin Set

Ein Hypergraph H ist ein Paar (F, S) wobei F eine nicht-leere endliche Menge ist und S eine Familie an Untermengen von F. Die Elemente von F sind Knoten und die Elemente von S sind Hyperedges.

5.4.2 Pruning von Unterbäumen

Manchmal ist es "günstig" den ganzen Baum zu untersuchen. Aber sonst sollte man den Suchraum verkleinern. Bei jedem Knoten wird dann beim Pruning die entscheidung getroffen, ob der Unterbaum behalten und durchsucht wird, oder verworfen wird.

5.4.3 Pruning von Unterbäumen die definitiv unnütz sind

- Pure Feasibility Problem: Ein Unterbaum im Entscheidungsbaum ist defintiv unnütz, wenn
  - der Unterbaum nicht zulässig ist
  - oder es mindestens eine zulässige Lösung außerhalb der Vereinigung zwischen Subtree und aller bisher abgeschnittenen Subtrees gibt
- Optimierungsproblem: Ein Unterbaum im Entscheidungsbaum ist defintiv unnütz, wenn
  - der Unterbaum nicht zulässig ist
  - oder der Subbaum unbounded ist (Dann ist das gesamte Problem unbounded)
  - oder es gibt mind. eine optimale Lösung (Falls diese Existiert) die außerhalb der Vereinigung zwischen Subtree und aller anderen zuvor abgeschnittenen Subtrees
  - oder wir wissen es gibt eine zulässige Lösung die besser ist als die optimale Lösung im Subtree

5.4.4 Unnötigkeit bestimmen

Bevor Subtrees abgeschnitten werden, muss bewiesen werden, dass dieser Unnötig ist. Bei NP-harten Problemen kann man nicht immer schnell bestimmen, dass es definitiv unnötig ist. $\Rightarrow$ man ist dann konservativ, und errechnet entweder das Ergebnis "definitiv Unnütz" oder "Keine Ahnung", um auf der sicheren Seite zu sein und nicht gute Branches zu prunen.

5.4.5 Branch-and-Bound

Annhame: wir kennen eine obere Schranke U auf dem optimalen Kostwert. Zusätzlich nehmen wir an, dass für jeden Knoten v ein lower Bound $l(v) \in \mathbb{R} \bigcup \{+\infty\}$ auf den Zielwerten aller zulässigen Lösungen im Subbaum mit Wurzel v berechnet werden kann. Wenn Lösungsmenge von v leer ist, dann setzt man $l(v) := +\infty$.

Der Subbaum mit Wurzel v ist unnütz, wenn ...

- $l(v) = +\infty$, weil Subbaum unzulässig (Pruning by infeasibility)
- $l(v) > U$ weil keine optimale Lösung im Subbaum ist (Pruning by bound)

- $l(v) = c(s)$ für eine gefundene zulässige Lösung s
  - wenn s Teil des Subbaums ist, dann wurde eine optimale Lösung für den gesamten Subbaum gefunden. Wenn $c(s) < U$ gilt, dann kann der Upper Bound $U := c(s)$ gesetzt werden.
  - sonst kann Subbaum eine Lösung enthalten mit gleichen Zielwert oder schlechter.
  - in beiden Fällen ist es Pruning by Optimality

5.4.6 Berechnung des Upper Bound Startwertes

Je kleiner der Upperbound, desto besser wird das Prunen. Um den Wert zu berechnen werden Heuristiken verwendet, und Lösungswerte können als Upperbound verwendet werden. Wenn man bei der Baumsuche eine zulässige Lösung findet s mit $c(s) < U$, dann kann man U mit $c(s)$ ersetzen. Es kann auch sein dass der Algorithmus welcher den Lower-Bound berechnet eine zulässige Lösung liefert, die ebenfalls verwendet werden kann um das Upper-Bound zu aktualisieren. Wenn eine gute Baumsuchstrategie gewählt wird, kann ein schlechter Upper-Bound schnell mit den Kosten einer besseren zulässigen Lösung ersetzt werden. Der Upper-Bound wird besser durch das durchführen der Suche.

5.4.7 Lower Bounds für Unterbäume

$\Rightarrow$ man benötigt eine untere Schranke l_u auf dem optimalen Zielvalue in der Untermenge.

Idee: man löst eine Entspannung der vorliegenden Instanz. Der Optimalzielwert der relaxierten Instanz ist die untere Schranke der originalen Instanz. (Ergibt jedoch nur Sinn, wenn das relaxierte Problem einfacher zu lösen ist, als das Originale Problem)

5.4.8 Relaxations

Man versucht also das Problem mit einem einfacheren Problem auszutauschen, wo der Optimalwert nicht größer ist, als das Originalproblem. Hierbei kann entweder die Menge der zulässigen Lösungen vergrößert werden oder die Zielfunktion durch eine Funktion die für jedes x den gleichen oder kleineren Wert hat.

Definition
Sei (ORIG) das folgende Problem:

$$opt^O = \min\{c(x)|x \in X \subseteq \mathbb{R}^n\}$$

Das Problem (REL) ist eine Relaxierung auf (ORIG) mit:

$$opt^R = \min\{f(x)|x \in T \subseteq \mathbb{R}^n\}$$

Dies aber nur wenn $X \subseteq T$ und $f(x) \leq c(x), \forall x \in X$
Eigenschaften einer Relaxierung

- Relaxierungen liefern untere Schranken: wenn REL eine Relaxierung von ORIG ist, dann gilt: $\text{opt}^R \leq \text{opt}^O$
- Wenn eine Relaxierung unzulässig ist, ist das Originalproblem unzulässig
- Relaxierungen können optimale Lösungen liefern. Sei $x*$ eine optimale Lösung für REL. Wenn $x* \in X$ und $f(x*) = c(x*)$, dann ist $x*$ eine optimale Lösung für ORIG.

5.4.9 Lagrangische Relaxierung

Für beliebigen Vektor λ eines Lagrangian Multiplikators, ist der Wert $L(\lambda)$ der Lagrange Funktion eine untere Schranke des Zielwertes $z(P)$ des Originalen Optimierungsproblems P.
Beweis: Daher dass $Ax = b$ für jede zulässige Lösung von P gilt, haben wir für beliebigen Vektor λ des Lagrange Multiplikators:

$$\min\{c^T x | Ax = b, x \in X\} = \min\{c^T * x + \lambda^T (Ax - b) | Ax = b, x \in X\}$$

Wenn $Ax = b$ nichtmehr beachtet wird, dann kann die Zielfunktion nicht einen höheren Wert erzeugen.

5.4.10 Constraint Satisfaction Problems

Ein Problem welches aus folgenden Dingen besteht:

- eine Menge an Variablen $X = \{x_1, x_2, ..., x_n\}$
- für jede Variable x_i eine endliche Menge D_i von möglichen Werten (der Domäne)
- eine Menge von Begrenzungen, die eingrenzen welche Werte die Variable annehmen kann

Eine Lösung zu einem CSP ist eine Abbildung von einem Wert der Domäne zu jeder Variable sodass jede Bengrenzung erfüllt ist. Somit ist CSP ein pures Zulässigkeitsproblem.

5.4.11 Backtracking

Goes a path in the search tree. If partial assignment violates any constraints, that sub-tree can be pruned. When it is pruned, it will go back the path it came from, until it reaches a node with alternatives.

Nachteile

- Trashing: Wiederholte Fehler für die gleichen Gründe
  Backtracking findet nicht die Gründe der Unzulässigkeiten. Trashing kann vermieden werden durch backjumping, zB durch das direkte Backtracking auf die Variable die den Fehler verursacht
- Backtracking führt Redundante Arbeit durch. Fehlerquellen werden erkannt aber nicht gespeichert. Dies kann vermieden werden durch Backchecking oder Backmarking.
- Backtracking erkennt Konflikte zu spät. Kann vermieden werden durch Konsistenztechniken um vorwärts zu kontrollieren.

5.4.12 Constraint (Hyper)graph

Für jedes CSP kann ein Constraint Hypergraph assoziiert werden. Die Knoten des Hypergraphen korrespondieren 1-zu-1 zu den Variablen. Jede Constraint formt eine Hyperkante, die Knoten die die Hyperkante bilden sind die, die in der Constraint vorkommen. Eine Constraint die nur eine Variable bindet, ist Unary. Wenn alle Constraints binär sind, dann ist der Hypergraph ein gerwöhnlicher Graph.

Knotenkonsistenz
Ein Knoten repräsentiert eine Variable X in einem Constraint Graph sind Knotenkonsistent, genau dann wenn für jeden Wert x in der aktuellen Domäne D_X von X, jeder unary Constraint auf X erfüllt ist.
Eine CSP ist Knotenkonsistent, wenn alle Variablen Knotenkonsistent sind. Wenn ein Wert nicht konsistent ist, kann dieser von der Domäne entfernt werden.

Arc-Consistency
Seien X, Y zwei Variablen welche in einer Binären Constraint vorkommen. Wir sagen dass das geordnete Paar (X, Y) sind Arc-Consistent, genau dann wenn für jeden Wert x in der aktuellen Domäne von D_X ein Wert y in der Domäne von Y sodass $X = x$ und $Y = y$ sind permitted durch die binäre Constraint zwischen X und Y.
Eine CSP ist Arc-Consistent wenn jede Arc (X, Y) in der Constraint Graph Arc-Consistent sind.

Arc-Consistency erreichen
Eine Arc kann Konsistent gemacht werden, indem die Werte von der Domäne von X gelöscht werden, für die es Kein Wert in der Domäne Y gibt, sodass die binäre Constraint zwischen X und Y erfüllt sind. Um Arc-Konsistenz generell zu erreichen, ist es notwendig diese Reduktion wiederholt durchzuführen solang die Domäne von irgendeiner Variable sich ändert.

Sind Arc-Consistencies ausreichend um zu zeigen, dass eine Lösung existiert?

Nein.

5.4.13 Constraing Propagation

Konsistenztechniken können den suchraum verkleinern. Solche Schemata werden look-ahead Strategien genannt und Sie sind basiert auf die Idee den Suchraum zu reduzieren durch Konstante Propagation.

5.4.14 Forward checking

Einfachste Art um Zukünftige Konflikte zu vermeiden, ist Forward Checking. Es führt Arc-Consistency zwischen Paaren von noch nicht instanziierten Variablen durch und einer weiteren instanziierten Variable. ...

5.4.15 Full Look Ahead

In Full Look Ahead werden die Constraints gecekecht zwischen der zukünftigen Variable und der aktuellen Variable.

5.4.16 Variable Ordering

Es existieren Heuristiken um die Variablen Reihenfolge zu entscheiden.

- Bevorzuge Variablen mit kleineren Domänen. Rationale: Um erfolgreich zu sein, versuche da wo du am wahrscheinlichsten Fehlschlägst. Wenn Verlust unvermeidbar ist, und dies so früh wie möglich entdeckt wird, desto besser.
- Wenn ein Unentschieden vortritt, wird die Variable bevorzugt die mehr constraints zu den instanziierten Variablen hat. Rationale: Zuerst die harten Fälle zuerst, diese können nur schwerer werden.

5.5 Dynamische Programmierung

DP ist eine exakte Optimierungsmethode, welche das Problem löst, indem die Lösungen von Subproblemen zusammengesetzt werden. Ein DP Algorithmus löst jedes Subproblem einmal und speichert diese in einer Tabelle, wobei vermieden wird, Lösungen zu berechnen zu Problemen die bereits gelöst wurden. In Kontrast, würde ein divide-and-conquer die Probleme lösen die bereits gelöst wurden (Dramatische Laufzeit). Ziel ist also Teilprobleme nur einmal zu berechnen.

5.5.1 DP-Ansatz designen

Man fängt unten an bei den kleinsten Teilproblemen und löst diese. Dann die zweitkleinsten Teilprobleme werden mithilfe der kleinsten Teilprobleme gelöst. Und so wird Schritt für Schritt,

Bottom-Up eine Lösung gebaut.
Es gibt zwei "Zutaten" um Dynamic Programming gescheit anzuwenden. Optimale Substrukturen, also dass eine optimale Lösung für ein Problem die optimalen Lösungen für Subprobleme beinhaltet. Und wenn sich die Teilprobleme überlappen, dann ergibt es Sinn DP zu nutzen. Wenn die Teilprobleme disjunkt sind, macht man einfach divide and conquer und man ist fertig. Da wo die sich überlappen, werden die selben Probleme gelöst, wenn man das rekursiv macht.

5.6 Greedy Algorithmen

Ein kurzsichtiger Algorithmus. Es wird ein Pfad in einem Suchbaum gefolgt, und bei jedem Knoten wird der Pfad gewählt, der zum aktuellen Zeitpunkt am vielversprechendsten ist. Greedy ist also eine simple Art von Backtracking, ohne Backtracking zu machen.

5.6.1 Unabhängigkeitssysteme

Sei E eine endliche Menge und $F \subseteq 2^E$.
Ein Mengensystem (E, F) ist ein Unabhängigkeitssystem, wenn die leere Menge in F enthalten ist und es gilt: $X \subseteq Y$ und $Y \in F$, dann $X \in F$. Die Elemente von F sind Unabhängig, und die Elemente der Potenzmenge von E ohne F sind abhängig. Eine Basis ist eine maximale Unabhängigkeitsmenge.
Beispiele für Optimierungsprobleme für Unabhängigkeitssysteme:

- Maximum Weight Stable Set Problem
- TSP
- Shortest Path Problem

Best-In-Greedy: Maximierungsproblem

Worst-Out-Greedy: Minimierungsproblem

$\Rightarrow$ Die Lösungen die von Greedy geliefert werden, können recht schlecht sein. Lokale Suchen liefern definitiv Optimale Lösungen bei exakten Nachbarschaften.

5.7 Matroiden

Ein Unabhängigkeitssystem ist ein Matroid wenn $X, Y \in F$ und $|X| > |Y|$, dann gibt es ein $x \in X \backslash Y$ mit $Y \cup \{x\} \in F$.

Matroiden sind allgemeine Matrixstrukturen.

5.7.1 Charakterisierung von Matroiden

Sei (E, F) ein Unabhängigkeitssystem. Dann sind folgende Aussagen equivalent.

1. wenn $X, Y \in F$ und $|X| > |Y|$, dann gibt es ein $x \in X \backslash Y$ mit $Y \cup \{x\} \in F$.
2. wenn $X, Y \in F$ und $|X| = |Y| + 1$, dann gibt es ein $x \in X \backslash Y$ mit $Y \cup \{x\} \in F$.
3. für alle $X \subseteq E$ gilt: alle Basen von X haben die selbe Kardinalität.

5.7.2 Matroid und Greedy

Sei (E, F) ein Unabhängigkeitssystem. Dies ist ein matroid wenn der Best-In-Greedy Algorithmus eine optimale Lösung für das Maximierungsproblem (E, F, c) für alle Kostenfunktionen $c : E \to \mathbb{R}$.

6 Approximation Algorithmen

6.1 Absolute Approximation

Ein polynomieller Algorithmus A für ein Optimierungsproblem P ist ein absoluter Approximationsalgorithmus falls eine Konstante k sodass für alle Instanzen I von P gilt: $|App(I) - Opt(I)| \leq k$.

$\Rightarrow$ Absolute Approximationsalgorithmen sind nur für ein paar NP-harte Probleme bekannt.

6.1.1 k-Faktor Approximations

Sei P ein Optimierungsproblem und $k \geq 1$ eine Konstante. Ein k-Faktor Approximationsalgorithmus für P ist ein polynomieller Algorithmus A für P sodass: $\frac{1}{k} * Opt(I) \leq App(I) \leq k * Opt(I)$. Man sagt auch A hat eine Leistungsratio k.