Advanced Problem Solving and Search

Henrik Tscherny

14. Juli 2022

Inhaltsverzeichnis

1 Local-Search

Local-Search has basicly 4 steps:

1. pick solution from the search space
2. evaluate that solution
3. pick a better solution in the neighbourhood if possible
4. repeat until no better solution can be found

1.1 Hill-climbing

- start from any initial point, =*a*
- choose a point in the neighbourhood of *a*, =*b*
- check if value of *a* is better then value of *b*
  - if so *a*=*b*
  - else choose different point *b*
- if search space (N(a)) is exhausted return

iterated hill-climbing:
basic hill-climbing but you terminate after n steps, good if function is unbounded and has no min/max

1.2 Examples

GSAT

1. randomly assign *True* or *False* to each Variable
2. flip one variable assignment
   - if SAT: return
   - else: continue with 2 till MAX-FLIPS is reached
3. continue with 1 till MAX-TRIES is reached

1.3 Simulated Annealing/Stochastic Hill-Climbing

Improvement to Local-Search to skip over local min/max through new parameter **temperature**

- points in the neighbourhood are selected probabilisticly
- probability depends on the value difference of the current and the neighbouring point as well as the temperature
- higher temperature means less impact of the value difference $\rightarrow$ search is more random
- formula for probability: $\frac{1}{1+e^{\frac{eval(v_c)-eval(v_n)}{T}}}$

- newly selected points can be worse then previous points (so that we can skip local optima)
- for **stochastic hill-climbing** the temperature remains constant
- for **simulated annealing** the temperature decreases over time making it more probable that new points are accepted over time
- also for **simulated annealing** new better valued points are always chosen

1.4 Tabu-Search

- Using a Memory to search new locations in the search space
- recently examined locations are ignored for a certain time period (e.g. five iterations)
- if some solution is much better then the solution before, the tabu can be overridden (**aspiration criteria**)
- long-term memory like **frequency based memory** can be used to chose steps when every next solution is worse then the solution before
$\rightarrow$ 20 out of 40 times we went right, so let's move right again
- tabu-search moves to worse locations only if stuck in a local min/max

2 Answer-Set Programming

- declarative problem solving approach
- modeling language
- allows solving problems in *NP* and NP^{NP}

2.1 Normal Logic Program

- $a_0 \leftarrow a_1, ..., a_m, not\ a_{m+1}, ..., not\ a_n$
  - head
  - body$^+$
  - body$^-$
  - body = body$^+$ $\cup$ body$^-$

- if body$^-$ = $\emptyset$ its called a *positive program*
- a set of Atoms **A** is *closed under* a positive program P iff the Head and body$^+$ only contains elements of **A**
  $\rightarrow$ X corresponds to a model of P
- the *smallest* set of Atoms closed under a positive program P is called $Cn(P)$
  $\rightarrow Cn(P)$ also corresponds to the smallest model of P
- $Cn(P)$ is a *stable model* of P
- *definitive clauses/positive rules*: $a_0 \vee \wedge a_1 ... \vee \wedge a_m$
  (exactly one positive atom in DNF)
- *Horn clauses*: clauses with **at most** one positive atom
  $\rightarrow$ every definite clause is a horn clause

Gelfond-Lifschitz-Reduct $\mathrm{P}^{\mathbf{X}}$

- $P^X = \{head(r) \rightarrow body^+ \,|\, r \in P$ and $body^- \cap X = \emptyset\}$
  $\rightarrow$ body$^-$ does not contains elements of X
- a set **X** of Atoms is a *stable model* of **P**, if $Cn(P^X) = X$
  - delete each rule having *not a* in its body with $a \in X$
  - delete negative atoms *not a* from the remaining rules

Variables in Logic Programs

- let *P* be a logic program with rules *r*
- $\mathcal{T}$: set of variable-free terms (Herbrand universe)
- $\mathcal{A}$: set of atoms constructable from $\mathcal{T}$ (Herbrand base)
- $ground(r) = \{r\theta \,|\, \theta : var(r) \rightarrow \mathcal{T}, var(r\theta = \emptyset\}$
- $ground(P) = \bigcup_{r \in P} ground(r) \Rightarrow$ *ground instantiation*
  Ground(P) is the Set of the facts and all the rules of P with each variable replaced by an element of $\mathcal{T}$, for all possible choices of elements from $\mathcal{T}$

Syntax

	true	false	if	and	or	iff	default negation	classical negation
source code			:-	,	|		not	-
logic program			$\leftarrow$	,	;		*not*	$\neg$
formula	$\top$	$\bot$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\sim$	$\neg$

default negation vs. classical negation

default negation	classical negation
only negates if there is evidence that sth. is not true $\rightarrow$ close-world assumption can be used to express non-monotonicity $\rightarrow$ more knowledge can lead to less true statements (knowledge can be 'revoked')	False is not the same as Unknown Nothing can be inferred about an object $\rightarrow$ open-word assumption more statements lead to more knowledge $\rightarrow$ monotonicity

Complexity:

	X is a stable model of P	a is in the stable model of P
positive normal logic program	P-complete	P-complete
normal logic program	P-complete	NP-complete
normal lp w/ optimization statements	co-NP-complete	Δ_2^P-complete
positive disjunctive lp	co-NP-complete	NP^{NP}-complete
disjunctive lp	co-NP-complete	NP^{NP}-complete
disjunctive lp w/ optimization statements	co-NP^{NP}-complete	Δ_3^P-complete
propositional theory	co-NP-complete	NP^{NP}-complete

3 Constraint Satisfaction Problem

$\mathcal{C} = \langle X, D, C \rangle$, with:

- X... Variables
- D... Domains for each Variable
  a domain contains a set of allowed values for each Variable (can be finite or infinite, can be discrete or continuous)
- C... Constraints for Variable values
  a tuple $\langle$scopre, rel$\rangle$
  scope is a tuple of constraint variables
  rel defines the possible values (can be a list or an expression etc.)
- other then in regular search in CSP its possible to cut large portions from search space at once using constraint $\rightarrow$ **constraint propagation**
  this can be done in combination with a search or as a pre-processing step
  - Each variable becomes a node
  - Each binary constraint becomes an arc
  - enforcing local consistency:

* Node consistency: all values in the domain of a Variable satisfy the unary constraints of that Variable
  you can remove elements from the domain that do not satisfy the constrain
* Arc consistency: all values from a domain for a variable satisfy the variables binary constraints. Two Variables are arc consistent if there there is a value in each Variables respective domain s.d. the binary relation is satisfied. If this hold for all variables the CSP is arc consistent
* Path consistent:

3.1 Example

Coloring of a Map s.d. no two adjacent countries are the same color

- $X =$
  $\{Germany, Netherlands, Belgium, Luxembourg, France, Switzerland, Lichtenstein, Austria, CzechRepublic, Poland, Denmark\}$
- $D = \{Red, Green, Blue, Yellow\}$
- $C =$
  adjacent countries have different colors (may be expressed in an formal language)
- a Solution would be a color assignment for each country s.d. all constraints are satisfied

3.2 Search strategies

- **Standard search formulation (incremental)**
  - initial state: $\emptyset$
  - states are defined by the values assigned so far
  - step: assign a value to an unassigned variable, (as well as checking for conflicts)
  - goal: current assignment is complete
  - $\Rightarrow$ every solution is at depth *n* with *n* Variables
  - $\Rightarrow$ branching factor $b = (n - l)d$ at depth l $\rightarrow n!d^n$ leaves
- **Backtracking search**

- variable assignments are commutative $\rightarrow$ it doesnt matter if we assign X before Y or Y before X
- branching factor $b = d$ with d^n leaves
- DFS for CSPs with single-variable assignment

4 Evolutionary Algorithms

- instead of modifying a existing solution we now use random variation to search for better solutions in parallel.
- based on natural selection
- each of the individuals within a population can be evaluated by a fitness function
  $\rightarrow$ fitter individual win over less fit competitors
- surviving individuals act as seed for the next population
  $\rightarrow$ genes are passed on
- recombination and mutation allows for new and potentially fitter individuals
- after a certain time the increase in fitness stagnates

5 Trees

- Many CSPs can be solved in P-Time if the problems treewidth is small
- Solving a bounded width problem includes two steps
  1. generate a (hyper)tree decomposition with small width
  2. solve the problem based on the decomposition with dynamic programming
- *Idea*: decompose main problem into sub-problems of smaller size
- if the constraint Graph to a corresponding CSP has no loops the CSP can be solved in $\mathbf{O(nd^2)}$ time
  (which is much smaller then the usual $O(d^n)$)
- *constraint graph*: Nodes... Variables, Arcs...Constraints

5.1 Tree Decompositions

- $G = (V, E)$
- *tree decomposition* (T, χ)
  - Tree: $T = (I, F)$, I...Nodes, F...Edges
  - $\chi = \{\chi_i : i \in I\}$ with:
    * $\bigcup_{i \in I} \chi_i = V$
    * $\forall (v, w) \in E \exists i \in I : v \in \chi_i$ and $w \in \chi_i$
    * $\forall i, j, k \in I :$ if j is on the path from i to k in T, then $\chi_i \cap \chi_k \subseteq \chi_j$
- width of a tree decomposition: $\max_{i \in I} |\chi_i| - 1$
- treewidth of a graph G is denoted by $\mathbf{tw}(\mathbf{G})$, its the minimum width over all possible tree decompositions of G
- finding tw(G) is **NP-hard**