SAT-Based Planning

• Using Propositional SAT-Solvers to Search for Plans
Literature

The General Idea
• idea: transform planning problem into other problem for which efficient solvers are known
• approach here:
  • transform planning problem into propositional satisfiability problem (SAT)
  • solve transformed problem using (efficient) SAT solver, e.g. GSAT
  • extract a solution to the planning problem from the solution to transformed problem

The General Idea
• idea: transform planning problem into other problem for which efficient solvers are known
• approach here:
  • transform planning problem into propositional satisfiability problem (SAT)
    • SAT: problem of determining whether a propositional formula is satisfiable (theorem proving)
  • solve transformed problem using (efficient) SAT solver, e.g. GSAT
  • extract a solution to the planning problem from the solution to transformed problem
• main advantage: exploit recent results in SAT solver for efficient planning
Overview

- **Encoding Planning Problems as Satisfiability Problems (SAT)**
  - now: how to encode a planning problem into SAT problem that has a model iff the planning problem has a solution
- **Efficient SAT Solving Algorithms**
Encoding a Planning Problem

- aim: encode a propositional planning problem $\mathcal{P}=(\Sigma, s_i, g)$ into a propositional formula $\Phi$ such that:
  - $\mathcal{P}$ has a solution if and only if $\Phi$ is satisfiable, and
  - every model $\mu$ of $\Phi$ corresponds to a solution plan $\pi$ of $\mathcal{P}$.

- key elements to encode:
  - world states
  - state-transitions (actions)

- model: assignment of truth values to propositions (atoms) in the formula such that formula evaluates to true

- formula is satisfiable if it has a model (if such an assignment exists)

- key elements to encode:
  - world states
  - state-transitions (actions)

- note: style of encoding determines length of formula and thus difficulty of SAT problem
Example: Simplified DWR Problem

- same problem as for Graphplan
- [figure]
- initial state: as shown
- robots can load and unload autonomously
- locations may contain unlimited number of robots and containers
- problem: swap locations of containers
Simplified DWR Problem: State Proposition Symbols

- **robots:**
  - \( r1 \) and \( r2 \): \( \text{at(robr,loc1)} \) and \( \text{at(robr,loc2)} \)
  - \( q1 \) and \( q2 \): \( \text{at(robq,loc1)} \) and \( \text{at(robq,loc2)} \)
  - \( ur \) and \( uq \): \( \text{unloaded(robr)} \) and \( \text{unloaded(robq)} \)

- **containers:**
  - \( a1 \), \( a2 \), \( ar \), and \( aq \): \( \text{in(conta,loc1)} \), \( \text{in(conta,loc2)} \), \( \text{loaded(conta,robr)} \), and \( \text{loaded(conta,robq)} \)
  - \( b1 \), \( b2 \), \( br \), and \( bq \): \( \text{in(contb,loc1)} \), \( \text{in(contb,loc2)} \), \( \text{loaded(contb,robr)} \), and \( \text{loaded(contb,robq)} \)

- initial state: \{\( r1 \), \( q2 \), \( a1 \), \( b2 \), \( ur \), \( uq \}\}
Encoding World States

- use conjunction of propositions that hold in the state
- example:
  - initial state: \( \{r_1, q_2, a_1, b_2, ur, uq\} \)
  - encoding: \( r_1 \land q_2 \land a_1 \land b_2 \land ur \land uq \)
  - model: \( \{r_1 \leftarrow \text{true}, q_2 \leftarrow \text{true}, a_1 \leftarrow \text{true}, b_2 \leftarrow \text{true}, ur \leftarrow \text{true}, uq \leftarrow \text{true}\} \)
Intended vs. Unintended Models

- possible models:
  - intended model: \{r1\leftarrow true, r2\leftarrow false, q1\leftarrow false, q2\leftarrow true, ur\leftarrow true, uq\leftarrow true, a1\leftarrow true, a2\leftarrow false, ar\leftarrow false, aq\leftarrow false, b1\leftarrow false, b2\leftarrow true, br\leftarrow false, bq \leftarrow false\}
  - unintended model: \{r1\leftarrow true, r2\leftarrow true, q1\leftarrow false, q2\leftarrow true, ur\leftarrow true, uq\leftarrow true, a1\leftarrow true, a2\leftarrow false, ar\leftarrow true, aq\leftarrow false, b1\leftarrow false, b2\leftarrow true, br\leftarrow false, bq \leftarrow false\}
  - encoding: add negated propositions not in state
    - example: \(r1 \land \neg r2 \land \neg q1 \land q2 \land ur \land uq \land a1 \land \neg a2 \land \neg ar \land \neg aq \land \neg b1 \land b2 \land \neg br \land \neg bq\)

- both models make formula from previous slide true
- differences are underlined
  - unintended model has container a in location 1 and on robot r (may be possible, but unintended)
  - unintended model has robot r in location 1 and 2 at the same time
Encoding the Set of Goal States

- goal: defined as set of states
  - example:
    - swap the containers
    - all states in which $a_2$ and $b_1$ are true
- propositional formula can encode multiple states:
  - example: $a_2 \land b_1$ ($2^{12}$ possible models)
  - use disjunctions for other types of goals

- note: every state can be described in this way, allowing for ambiguity
Simplified DWR Problem: Action Symbols

- move actions:
  - Mr12: move(robr,loc1,loc2), Mr21: move(robr,loc2,loc1), Mq12: move(robq,loc1,loc2), Mq21: move(robq,loc2,loc1)

- load actions:
  - Lar1: load(conta,robr,loc1); Lar2, Laq1, Laq2, Lar1, Lbr2, Lbq1, and Lbq2 correspondingly

- unload actions:
  - Uar1: unload(conta,robr,loc1); Uar2, Uaq1, Uaq2, Uar1, Ubr2, Ubq1, and Ubq2 correspondingly

- 20 action symbols
Extended State Propositions

- **figure**
  - formulas describe state transition when applying Mr12 in state $s_1$, resulting in $s_2$
  - **state transition**: $\gamma(s_1, Mr12) = s_2$ where:
    - $s_1$ described by $r1 \land \neg r2$ and
    - $s_2$ described by $\neg r1 \land r2$
  - **problem**: $r1 \land \neg r2 \land r1 \land r2$ has no model
  - **idea**: extend propositions with state index
    - **example**: $r1_1 \land \neg r2_1 \land \neg r1_2 \land r2_2$
    - **model**: $\{r1_1 \leftarrow \text{true}, r2_1 \leftarrow \text{false}, r1_2 \leftarrow \text{false}, r2_2 \leftarrow \text{true}\}$

- **Extended State Propositions**

  - state transition: $\gamma(s, Mr12) = s_2$ where:
    - $s_1$ described by $r1 \land \neg r2$ and
    - $s_2$ described by $\neg r1 \land r2$
  - **problem**: $r1 \land \neg r2 \land r1 \land r2$ has no model
  - **idea**: extend propositions with state index
    - **example**: $r1_1 \land \neg r2_1 \land \neg r1_2 \land r2_2$
    - **model**: $\{r1_1 \leftarrow \text{true}, r2_1 \leftarrow \text{false}, r1_2 \leftarrow \text{false}, r2_2 \leftarrow \text{true}\}$

- **Extended State Propositions**

  - state transition: $\gamma(s, Mr12) = s_2$ where:
    - $s_1$ described by $r1 \land \neg r2$ and
    - $s_2$ described by $\neg r1 \land r2$
  - **problem**: $r1 \land \neg r2 \land r1 \land r2$ has no model
  - **idea**: extend propositions with state index
    - **example**: $r1_1 \land \neg r2_1 \land \neg r1_2 \land r2_2$
    - **model**: $\{r1_1 \leftarrow \text{true}, r2_1 \leftarrow \text{false}, r1_2 \leftarrow \text{false}, r2_2 \leftarrow \text{true}\}$
Extended Action Propositions

- use same mechanism to describe actions applied in different states:
  - example: Mr12_1: move robot $r$ from location 1 to location 2 in state $s_2$
  - action encoding: $\text{Mr12}_1 \Rightarrow (r_1 \land \neg r_2 \land \neg r_1 \land r_2)$

- Mr12_1: state index refers to state before that action and has same number; different for planning graph

- action encoding: $\text{Mr12}_1 \Rightarrow (r_1 \land \neg r_2 \land \neg r_1 \land r_2)$
  - model: action must be true; rest remains as before

- [figure]

- note:
  - encoding based on same ideas as situation calculus, but propositional
  - encoding of actions as propositional formulas (propositional representation: actions as set-theoretic operators)
Bounded Planning Problems

• encoding in two steps:
  - bounded planning problem: for a given planning problem $\mathcal{P}=(\Sigma,s_i,g)$ find a solution plan of a fixed length $n$
  - encode the bounded planning problem into a satisfiability problem
    - state propositions with index 0 … $n$
    - action propositions with index 0 … $n-1$

• idea: extend length of bounded planning problem until a plan is found
  - e.g. use binary search on plan length

• encode the bounded planning problem into a satisfiability problem
  - state propositions with index 0 … $n$
  - action propositions with index 0 … $n-1$

• note: if a plan of length $k < n$ exists then it can be extended to a plan of length $n$ with no-op actions
Encoding Bounded Planning Problems

- conjunction of formulas describing:
  - the initial state
  - the goal states
  - actions (applicability and effects)
  - frame axioms
  - one action at a time

- different types of encodings known: differ in resulting formula length, roughly equivalent to complexity of SAT problem
Encoding Initial and Goal States

- Let $F$ be the set of state propositions (fluents). Let $f \in F$.

- initial state:
  - $\bigwedge_{f \in si} f_0 \land \bigwedge_{f \in si} \neg f_0$

- goal states:
  - $\bigwedge_{f \in g^+} f_n \land \bigwedge_{f \in g^-} \neg f_n$
Encoding Actions

• Let $A$ be the set of action propositions. Let $a \in A$.

• for $0 \leq i \leq n-1$:
  • $a_i \Rightarrow (\bigwedge_{f \in \text{precond}(a)} f_i \land$
    $\bigwedge_{f \in \text{effects}^+(a)} f_{i+1} \land$
    $\bigwedge_{f \in \text{effects}^-(a)} \neg f_{i+1})$

if action $a$ is performed in step $i$ then:
  • all preconditions must have held before and
  • all positive effects will hold after and
  • all negative effects will not hold after
Encoding Frame Axioms

- use explanation closure axioms for more compact SAT problem

- for $0 \leq i \leq n-1$:
  - $(f_i \land \neg f_{i+1}) \Rightarrow (\forall a \in A \land \forall \text{effects}(a) \ a_i) \land$
  - $(\neg f_i \land f_{i+1}) \Rightarrow (\forall a \in A \land \forall \text{effects}(a) \ a_i)$

$\neg f_i \land f_{i+1}$
Encoding Exclusion Axioms

- allow only exactly one action at each step

- for $0 \leq i \leq n-1$ and $a\neq a'$, $a,a' \in A$:
  - $\neg a_i \lor \neg a'_i$

Unlike Graphplan: only one action in each step
Overview

- **Encoding Planning Problems as Satisfiability Problems (SAT)**
  - just done: how to encode a planning problem into SAT problem that has a model iff the planning problem has a solution

- **Efficient SAT Solving Algorithms**
  - now: efficient algorithms for SAT solving (very quick overview)
**Generic SAT Problem**

- given: set of $m$ propositional formulas: 
  \[ \{F_1 \ldots F_m\} \]
  - containing $n$ proposition symbols: $P_1 \ldots P_n$
- find: an interpretation $I$
  - that assigns truth values (T, F) to $P_1 \ldots P_n$, i.e. $I(F_j) = T$ or $I(F_j) = F$, and
  - under which all the formulas evaluate to T, i.e. $I(F_1 \land \ldots \land F_m) = T$

**Generic SAT Problem**

- given: set of $m$ propositional formulas: \{\(F_1 \ldots F_m\)\}
  - or one formula that is the conjunction of the formulas in the set
- containing $n$ proposition symbols: $P_1 \ldots P_n$
- find: an interpretation $I$
  - that assigns truth values (T, F) to $P_1 \ldots P_n$, i.e. $I(F_j) = T$ or $I(F_j) = F$, and
  - under which all the formulas evaluate to T, i.e. $I(F_1 \land \ldots \land F_m) = T$
Conjunctive Normal Form

- formula $F$ is in conjunctive normal form (CNF) iff:
  - $F$ has the form $F_1 \land \ldots \land F_n$ and
  - each $F_i$, $i \in 1\ldots n$, is a disjunction of literals

- Proposition: Let $F$ be a propositional formula. Then there exists a propositional formula $F'$ in CNF such that:
  - $F$ and $F'$ are equivalent, i.e.
  - for every interpretation $I$, $I(F) = I(F')$
Transformation into CNF

- eliminate implications:
  \[ F \leftrightarrow G = F \rightarrow G \land G \rightarrow F \]
  \[ F \rightarrow G = \neg F \lor G \]

- bring negations before atoms:
  \[ \neg (F \lor G) = \neg F \land \neg G \]
  \[ \neg (F \land G) = \neg F \lor \neg G \]
  \[ \neg \neg F = F \]

- apply distributive laws:
  \[ F \land (G \lor H) = (F \land G) \lor (F \land H) \]
  \[ F \lor (G \land H) = (F \lor G) \land (F \lor H) \]
SAT Solving Procedures

• systematic:
  • Davis-Putnam algorithm
    • extend partial assignment into complete assignment
    • sound and complete

• stochastic:
  • local search algorithms (GSAT, WalkSAT)
    • modify randomly chosen total assignment
    • sound, not complete, very fast
Local Search Algorithms

• basic principles:
  • keep only a single (complete) state in memory
  • generate only the neighbours of that state
  • keep one of the neighbours and discard others

• key features:
  • no search paths
  • neither systematic nor incremental

• key advantages:
  • use very little memory (constant amount)
  • find solutions in search spaces too large for systematic algorithms

Local Search Algorithms
• basic principles:
  • keep only a single (complete) state in memory
  • generate only the neighbours of that state
  • keep one of the neighbours and discard others

• key features:
  • no search paths
  • neither systematic nor incremental

• key advantages:
  • use very little memory (constant amount)
  • find solutions in search spaces too large for systematic algorithms
Random-Restart Hill-Climbing

• method:
  • conduct a series of hill-climbing searches from randomly generated initial states
  • stop when a goal is found

• analysis:
  • complete with probability approaching 1
  • requires $1/p$ restarts where $p$ is the probability of success
    $(1$ success $+ 1/p-1$ failures)

Random-Restart Hill-Climbing

• method:
  • must be able to generate random states; therefore usually complete state formulation
  • conduct a series of hill-climbing searches from randomly generated initial states
  • stop when a goal is found

• analysis:
  • complete with probability approaching 1
  • requires $1/p$ restarts where $p$ is the probability of success
    $(1$ success $+ 1/p-1$ failures)
  • no guarantee that algorithm will terminate; hence no time complexity
  • space complexity: $b$, the branching factor
  • completeness: eventually, it will generate the global maximum
  • optimality: same as completeness
Hill Climbing: getBestSuccessors

- getBestSuccessors(i, clauses)
  - given an interpretation and a set of clauses, find best “neighbours”
  - tc ← -1; succs ← {}
  - for every proposition p in i
    - i’ ← i.flipValueOf(p)
    - n ← number of clauses true under i’
    - if n > tc then tc ← n; succs ← {}
    - if n = tc then succs ← succs + i’
  - return succs
GSAT: Pseudo Code

- function **GSAT**(clauses)
  - returns a model or failure
  - 
    - *props* \(\leftarrow\) clauses.getPropositions()
    - *loop* at most MAXLOOP times
      - *i* \(\leftarrow\) randomInterpretation(props)
      - *while* not clauses.evaluate(i) *do*
        - *succs* \(\leftarrow\) getBestSuccessors(i, clauses)
        - *i* \(\leftarrow\) succs.selectOne()
      - *if* clauses.evaluate(i) *return* *i*
    - *return* unknown

- variants:
  - break inner loop when no improvement is made
  - allow only a limited number of level steps
GSAT Evaluation

• experimental results:
  • solved every problem correctly that Davis-Putnam could solve, only much faster
  • begins to return “unknown” on problems orders of magnitude larger than Davis-Putnam can solve

• analysis:
  • problems with many local maxima are difficult for GSAT
WalkSAT

- inspired by simulated annealing

- idea:
  - start with random interpretation
  - choose a random proposition to flip
  - accept if it represents an uphill or level move
  - otherwise accept it with probability $e^{-\delta/T(s)}$
    where:
    - $\delta = \text{decrease in number of true clauses under } i'$
    - $T(s) = \text{monotonically decreasing function from number of steps taken to temperature value}$
Overview

- Encoding Planning Problems as Satisfiability Problems (SAT)
- Efficient SAT Solving Algorithms

• Efficient SAT Solving Algorithms
  • just done: efficient algorithms for SAT solving (very quick overview)