Material


• Homework: Due Monday, May 2\textsuperscript{nd}.
  – Do: problems 5.1, 5.3, 5.5, 5.8, 5.9, 5.11

• Reminder: Midterm next Monday, May 2\textsuperscript{nd} (will cover up to and including Chapter 5, and Bacchus tutorial)
On the Midterm

• Closed book/closed notes
• Will cover all of book chapters 1-5 + Bacchhus tutorial (except for Bacchus chapter 4 on the Davis Putnam procedure)
• 1 hour 20 minutes
• No real programming questions (i.e., I probably won’t ask you to write code, or if I do they’ll be only very short program fragments that you can write in pseudocode).
• Mostly will test concepts, e.g.:
  – here is a question, express it as constraint satisfaction
  – what would the best search method for problem x be and why? Give the complexity for each of them to justify answer
  – Can you prove that a given CSP problem is (i,j) consistent
• Questions will be designed to be thought questions more so than plug-and-chug (if you are finding that you are writing pages of stuff, you are probably not doing the problem correctly).
• I’ll be on email for any questions you might have.
Outline For Today

• Finish Bacchus tutorial
  – note: some of this material slightly overlaps with chapter 5, but using more precise notation.
No-good reasoning

• What are no-goods good for?
  – For Intelligent Backtracking
    • logic can justify intelligent backtracking (e.g., to previous level or to much earlier in search tree)
  – For domain pruning
    • if we find a no-good \{1,2,4,V \leftarrow a\}, where V is a future variable, we can prune \(a\) from the domain of \(V\) at the current time (but we need to restore it when we return from level 4)
Generic Tree-Search Template

TreeSearch(level)
    if all variables are assigned
        current assignments is a solution, enumerate it
        if FINDALL
            set \{1, \ldots, level-2\} as a new no-good for assignment at level-1
        return (level-1)
    else
        return (0)

V := pickNextVariable()
BTL := level
for x ∈ CurrentDom[V]
    assign(V,x,level)
    if checkConsistent(V,x,level)
        propagateConstraints(V,x,level)
        BTL := TreeSearch(level+1)
    undo(V,x,level)
    if BTL < level
        return (BTL)

BTL := computeBackTrackLevel(V,level)
setNoGoodOfAssignmentatBTLlevel(BTL,V,level)
return (BTL)
Generic Tree-Search Template

- **FINDALL**: if we should find all solutions. Note that if we find a solution, it is not another solution so it becomes a no-good for the rest of the search.
- **pickNextVariable**: determines next variable to split on
  - this is important, can have large impact on search. “Fail First” principle later
  - In any case, picks variables with empty domains first to find deadends, and next picks variables with only one value in the domains
- **checkConsistent**: checks that newly assigned value is consistent
- **propagateConstraints**: propagate any constraints now available by the newly assigned variable (e.g., forward checking or arc consistency to future vars)
- **TreeSearch**: might want to backtrack below current level and if so, we backtrack back to that level right there, in inner loop
- **computeBacktrackLevel**: if all values of V failed, we might want to backtrack to level earlier than previous level
- **setNoGoodOfAssignmentatBTLevel**: if any no-goods can be computed, they are done so here.
Generic Backtracking

```
BT(level)
    if all variables are assigned
        current assignments is a solution, enumerate it
        if FINDALL
            return(level-1)
        else
            return(0)
    
V := pickNextVariable()
BTLevel := level
for x ∈ Dom[V]
    assign(V,x,level)
    if checkConsistent(V,x,level)
        BTLevel := BT(level+1)
        undo(V,x,level)
    if BTLevel < level // Only occurs when BTLevel == 0
        return(BTLevel)
    
return(level-1)
```
Generic Backjumping

\[ BJ(level) \]
\[
  \text{if all variables are assigned} \\
  \quad \text{current assignments is a solution, enumerate it} \\
  \quad \text{if \ FIND ALL} \\
  \quad \quad \text{maxBJLevel}[\text{level-1}] = \text{level-2} \\
  \quad \quad \text{return } \text{level-1} \\
  \quad \text{else} \\
  \quad \quad \text{return } (0) \\
\]

\[
V := \text{pickNextVariable}() \\
\text{BTLevel} := \text{level} \\
\text{maxBJLevel}[\text{level}] := 0 \\
\text{for } x \in \text{Dom}[V] \\
\quad \text{assign}(V,x,\text{level}) \\
\quad \text{if } (M := \text{checkConsistentBJ}(V,x,\text{level})) = \text{level} \\
\quad \quad \text{BTLevel} := BJ(\text{level+1}) \\
\quad \text{else} \\
\quad \quad \text{maxBJLevel}[\text{level}] = \max(\text{maxBJLevel}[\text{level}], M) \\
\quad \quad \text{undo}(V,x,\text{level}) \\
\quad \text{if } \text{BTLevel} < \text{level} \\
\quad \quad \text{return } (\text{BTLevel}) \\
\]

\[
\text{BTLevel} := \text{maxBJLevel}[\text{level}] \\
\text{maxBJLevel}[	ext{BTLevel}] := \max(\text{maxBJLevel}[\text{BTLevel}], \text{BTLevel-1}) \\
\quad \quad \text{// Note that this max is equal to BTLevel-1.} \\
\quad \text{return } (\text{BTLevel}) \\
\]

\[
\text{checkConsistencyBJ}(V,x,\text{level}) \\
\quad \text{for } i := 1 \text{ to level-1} \\
\quad \text{for all } C \text{ such that} \\
\quad \quad \text{1. } V \in \text{VarsOf}[C] \\
\quad \quad \text{2. All other variables of } C \text{ are assigned at level } i \text{ or above} \\
\quad \text{if } !\text{checkConstraint}(C) \quad \text{//Check against current assignment.} \\
\quad \quad \text{return } (i) \\
\quad \text{return } (\text{level}) \\
\]
Conflict-Directed Backjumping

```c
CBJ(level)
    if all variables are assigned
        current assignments is a solution, enumerate it
        if FINDALL
            NoGoodSet[level-1] := {1, ..., level-2}
            return level-1
        else
            return 0
    V := pickNextVariable()
    BTLevel := level
    NoGoodSet[level] := ∅
    for x ∈ Dom[V]
        assign(V, x, level)
        if (NoGood := checkConsistentCBJ(V, x, level)) == ∅
            BTLevel := CBJ(level+1)
        else
            NoGoodSet[level] = NoGoodSet[level] U (NoGood - {level})
            undo(V, x, level)
        if BTLevel < level
            return (BTLevel)
    BTLevel := max(NoGoodSet[level])
    NoGoodSet[BTLevel] := NoGoodSet[BTLevel] U (NoGoodSet[level] - {BTLevel})
    return (BTLevel)

checkConsistencyCBJ(V, x, level)
    for i := 1 to level-1
        for all C such that
            1. V ∈ VarsOf[C]
            2. All other variables of C are assigned at level i or above
            if !checkConstraint (C) // Check against current assignment.
                return (Levels VarsOf[C] were assigned)
    return (∅)
```
Conflict-Directed Backjumping

• Key features:
  – we store the no-good explicitly, stores dynamically updated union of no-goods associated with each variable (recall, no-goods are stored as levels currently assigned).
  – The explicit no-good is then used for backjumping.
  – some no-goods come from checking constraints (routine checkConsistentCBJ)
  – Question: how do we know that the constraint at the current level, if unsatisfied due to a cause say at level-n, will jump back to the appropriate previous level if a constraint at a lower level is also unsatisfied but say at level-m, where m<n?
Value Specific CBJ

\[
vSCBJ(\text{level})
\]
\[
\text{if all variables are assigned}
\]
\[
\text{current assignments is a solution, enumerate it}
\]
\[
\text{if FINDALL}
\]
\[
\text{NoGoodSet}[\text{VarAt}[\text{level-1}],\text{ValAt}[\text{level-1}]] := \{1, \ldots, \text{level-2}\}
\]
\[
\text{return level-1}
\]
\[
\text{else}
\]
\[
\text{return (0)}
\]
\[
\text{V := pickNextVariable()}
\]
\[
\text{BTLevel := level}
\]
\[
\text{for } x \in \text{Dom}[V]
\]
\[
\text{assign }(V,x,\text{level})
\]
\[
\text{if } (\text{NoGood := checkConsistentCBJ}(V,x,\text{level})) == \emptyset
\]
\[
\text{BTLevel := vSCBJ(\text{level+1})}
\]
\[
\text{else}
\]
\[
\text{NoGoodSet}[V,x] = \text{NoGood} - \{\text{level}\}
\]
\[
\text{undo}(V,x,\text{level})
\]
\[
\text{if BTLevel < level}
\]
\[
\text{return (BTLevel)}
\]
\[
\text{BTLevel := max}_{x \in \text{Dom}[V]} (\text{NoGoodSet}[V,x])
\]
\[
\text{C := Constraint between VarAt[BTLevel] and V}
\]
\[
\text{//C is the universal constraint if there is no constraint}
\]
\[
\text{//C, such that VarsOf } [C] = \{\text{VarAT}[\text{BTLevel}],V\},
\]
\[
\text{//between these two variables exists.}
\]
\[
\text{NoGoodSet}[\text{VarAt}[\text{BTLevel}],\text{ValAt}[\text{BTLevel}]] :=
\]
\[
\bigcup_{x : x \in \text{Dom}[V] \text{ and } C(\text{ValAt}[\text{BTLevel}],x) \text{NoGoodSet}[V,x]} - \{\text{BTLevel}\}
\]
\[
\text{return (BTLevel)}
\]
Value Specific CBJ

- Key things:
  - the no-good sets are again explicitly recorded, but now not only for each variable but also for each value.
  - We next do “constraint filtered unioning” of the no-goods

\[
\text{NoGoodSet}[\text{VarAt}[\text{BTLevel}], \text{ValAt}[\text{BTLevel}]] := \\
\bigcup_{x : x \in \text{Dom}[V]} \text{and } C(\text{ValAt}[\text{BTLevel}], x) \text{NoGoodSet}[V, x] - \{\text{BTLevel}\}
\]

- Ultimate goal is to do longer backjumps (if we backjump to the true cause, we will not do extra work).
- Cons: You can see that the no-good data structure is more complicated, and could become large. Backjumping also needs to manage the deletion of the no goods.
- Inherent tradeoff: ideally, we want to manage no-goods in a smart way, but the smarter the no-good processing, the more overhead we pay by processing no-goods. For some problems, this overhead can start costing more than the savings we get.
More advanced options

- Domain pruning
  - keep a current domain of values for each variable, prune out values we know are impossible, restore them when they become possible again.
  - This is reflected in the `Currentdom[V]` data structure which also must be updated as we prune (before the domain of each variable was fixed in advance).
    - this itself is a design issue. A quick for loop in C is fast if the domain is moderately sized, but iterators over a set are much slower. We must be sure that 1) the domain is big, and 2) we prune by quite a bit, or 3) the effects of domain pruning (e.g., sub-tree) is very large, before we resort to domain pruning.
Domain Pruning

```
vSCBJ+P(level)
  if all variables are assigned
    current assignments is a solution, enumerate it
    if FINDALL
      NoGoodSet[VarAt[level-1],ValAt[level-1]] := {1,...,level-2}
      prune(VarAt[level-1],ValAt[level-1],level-2)
      return level-1
    else
      return 0
  
  V := pickNextVariable()
  BTLevel := level
  for x ∈ Currentdom[V]
    assign(V,x,level)
    if (NoGood := checkConsistentCBJ(V,x,level)) == Ø
      BTLevel := vSCBJ+P(level+1)
    else
      NoGoodSet[V,x] = Nogood - {level}
      prune(V,x,max(NoGoodSet[V,x]))
    undo(V,x,level)
    if BTLevel < level
      return (BTLevel)

  BTLevel := max_{x∈Dom[V]}prunedLevel[V,x]
  C := Constraint between VarAt[BTLevel] and V
  // C is the universal constraint if no constraint
  // exists between these two variables

  NoGoodSet[VarAt[BTLevel],ValAt[BTLevel]] :=
  \bigcup_{x: x ∈ Dom[V] and C(ValAt[BTLevel],x)} NoGoodSet[V,x] - {BTLevel}
  prune(VarAt[BTLevel],ValAt[BTLevel],
        max(NoGoodSet[VarAt[BTLevel],ValAt[BTLevel]]))
  return (BTLevel)
```
More advanced options

- Constraint propagation
  - Use forward checking or AC-3 to for each sub-CSP prune from the domains of variables values that we know are not possible.
  - Other more advanced algorithms are:
    - MAC: Maintain Arc Consistency: It does (1,1)-consistency in each sub-CSP. Can be combined with CBJ.
    - GAC: generalized arc consistency (rather than waiting for constraint that has all but one value instantiated (FC) or two values instantiated (MAC), we can do more, to obtain (i,j)-consistency in the sub-CSP, but this can get expensive to maintain).
Dynamic Variable Orderings

- The `pickNextVar` routine need not always choose the next variable in all circumstances.
- In the most general case, we choose both the next variable and a value of the next variable at the same time, and each time we choose this, it can be different and might depend on anything that has previously happened during the search.
- This is shown on algorithm in next slide:
General Dynamic Variable order

```
FlexTreeSearch(level)
    if all variables are assigned
        current assignments is a solution, enumerate it
        if FINDALL
            {1,...,level-2} is a new no-good for assignment
            at level-1
            prune(VarAt[level-1], ValAt[level-1], level-2)
            return(level-1)
        else
            return(0)
    NEXT:
        (V,x) := pickNextVarVal()
        // V is unassigned, and x is from CurrentDom[V].
        // Furthermore,
        // 1. If there is a variable with an empty CurrentDom
        //     return such a variable and NIL as value x.
        // 2. Otherwise if there is a variable with a singleton
        //     CurrentDom, then return such a variable with x
        //     set to its single remaining value.

        if(x != NIL)
            assign(V,x, level)
            BTLevel := level
            if checkConsistent(V,x, level)
                propagateConstraints(V, x, level)
                BTLevel := FlexTreeSearch(level+1)
            undo(V,x, level)
            if BTLevel < level
                return(BTLevel)
            goto NEXT
            // We drop out here only if we have a variable with an empty domain.
            NoGood := ComputeNoGoodFromExhaustedVariable(V)
            BTLevel := max(NoGood)
            setNoGoodOfAssignmentatBTLevel(BTLevel, NoGood-{BTLevel})
            prune(VarAt[BTLevel], ValAt[BTLevel], max(NoGood-{BTLevel}))
            return(BTLevel)
```
Dynamic Variable order

- **Current Practice**
  - In graph coloring, a heuristic that works well is the “Brelaz heuristic”
    - first color vertex of maximum degree.
    - Next, select uncolored vertex of maximum saturation (maximum number of *different* colored vertices connected to it).
      - break ties by the degree of the vertex intersected with the uncolored nodes.
    - Maximizing number of different colored vertices puts most constraints on new variable choosen.
Dynamic Variable order

• Brelaz heuristic is essentially same as:
  – choose next variable with minimum remaining consistent values
  – break ties by number of constraints variable has with set of currently unassigned variables.

• This is example of the “fail first” principle
  – the best search order is one that minimizes the expected length (depth) of each branch (i.e., prefer short trees since search is exponential in the depth).
Other good sources