Checking for Solution Existence

- In some problems, we don’t care about a path, but about a configuration that has a desired property
- Instead of a goal, we have a target, which can be a set of states that satisfy some property

- We call the set of properties that legal solutions must obey constraints
- We call these problems constraint satisfaction problems (CSPs)
CSP Examples

• Satisfying curriculum/major requirements

• Sudoku

• Seating arrangements at a party

• LSAT Questions:
  http://www.thelsattrainer.com/sample-lsat-logic-games.html

CSPs

• Specifying CSPs
• One view: Search with special goal criteria
• CSP definition (general):
  – Variables \( X_1, \ldots, X_n \)
  – Variable \( X_i \) has domain \( D_i \)
  – Constraints \( C_1, \ldots, C_m \)
  – Solution: Each variable gets a value from its domain such that no constraints violated
• CSP examples...
  – http://www.csplib.org/
CSP Example

Graph coloring:

Western Australia (WA)
Northern Territory (NT)
Queensland (Q)
South Australia (SA)
New South Whales (NSW)
Tasmania (T)
Victoria (V)

Problem: Assign Red, Green and Blue so that no 2 adjacent regions have the same color. (3-coloring)

CSP as a Search Problem

- \( n \) variables \( X_1, \ldots, X_n \)
- Valid assignment: \( \{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}\}, \ 0 \leq k \leq n \), such that the values \( v_{i1}, \ldots, v_{ik} \) satisfy all constraints relating the variables \( X_{i1}, \ldots, X_{ik} \)
- Complete assignment: one where \( k = n \)
  [if all variable domains have size \( d \), there are \( O(d^n) \) complete assignments]
- States: valid assignments
- Initial state: empty assignment \( \{\} \), i.e. \( k = 0 \)
- Successor of a state:
  \( \{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}\} \rightarrow \{X_{i1} \leftarrow v_{i1}, \ldots, X_{ik} \leftarrow v_{ik}, X_{ik+1} \leftarrow v_{ik+1}\} \)
- Goal test: \( k = n \)
Backtracking Search

- Essentially a simplified depth-first algorithm using recursion

Backtracking Search

(3 variables)

Assignment = {}
Backtracking Search
(3 variables)

Assignment = \{(X_1, v_{11})\}

Backtracking Search
(3 variables)

Assignment = \{(X_1, v_{11}), (X_3, v_{31})\}
Backtracking Search
(3 variables)

Assumption: Assume that no value of $X_2$ leads to a valid assignment.

Assignment = $\{(X_1, v_{11}), (X_3, v_{31})\}$

Then, the search algorithm backtracks to the previous variable ($X_3$) and tries another value.

Assignment = $\{(X_1, v_{11}), (X_3, v_{32})\}$
Backtracking Search
(3 variables)

Assume again that no value of $X_2$ leads to a valid assignment.

The search algorithm backtracks to the previous variable ($X_3$) and tries another value. But assume that $X_3$ has only two possible values. The algorithm backtracks to $X_1$.

Assignment = \{(X_1, v_{11}), (X_3, v_{32})\}
Backtracking Search
(3 variables)

Assignment = \{(X_1, v_{12}), (X_2, v_{21})\}

The algorithm need not consider the variables in the same order in this sub-tree as in the other

Assignment = \{(X_1, v_{12}), (X_2, v_{21})\}
Backtracking Search
(3 variables)

Assignment = \{(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})\}

The algorithm need not consider the values of X_3 in the same order in this sub-tree

Assignment = \{(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})\}
Backtracking Search
(3 variables)

Assignment = \{(X_1, v_{12}), (X_2, v_{21}), (X_3, v_{32})\}

Since there are only three variables, the assignment is complete

Backtracking Algorithm

CSP-BACKTRACKING(A)

1. If assignment A is complete then return A
2. \(X \leftarrow\) select a variable not in A
3. \(D \leftarrow\) select an ordering on the domain of \(X\)
4. For each value \(v\) in \(D\) do
   a. Add \((X \leftarrow v)\) to A
   b. If A is valid then
      i. result \(\leftarrow\) CSP-BACKTRACKING(A)
      ii. If result \(\neq\) failure then return result
   c. Remove \((X \leftarrow v)\) from A
5. Return failure
Efficiency of CSP-Backtracking

CSP-BACKTRACKING(A)
1. If assignment A is complete then return A
2. X ← select a variable not in A
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to A
   b. If a is valid then
      i. result ← CSP-BACKTRACKING(A)
      ii. If result ≠ failure then return result
   c. Remove (X←v) from A
5. Return failure

Practical Efficiency of CSP Algorithms

• Fundamental trade off
  – Time spent ruling out bad/impossible choices
  – Time spent searching

• Try to find the sweet spot where you quickly rule out bad/unpromising choices
• Compare with sweet spot for heuristics in A*
CSP Example Revisited

Graph coloring:

Problem: Assign Red, Green and Blue so that no 2 adjacent regions have the same color. (3-coloring)

Example Contd.

- Variables: \{WA, NT, Q, SA, NSW, V, T\}
- Domains: \{R,G,B\}
- Constraints:
  - For WA – NT:\{(R,G), (R,B), (G,B), (G,R), (B,R), (B,G)\}
- We have a table for each adjacent pair
- Note: Many possible ways to express constraints
Forward Checking

• Idea: Assignments to variables immediately rule out certain assignments to other variables

• Remove illegal/invalid options from the domains of other variables

• You probably do this when you play Sudoku!


Forward Checking in Map Coloring

Constraint graph

WA  NT  Q  NSW  V  SA  T
RGB  RGB  RGB  RGB  RGB  RGB  RGB
Forward Checking in Map Coloring

Forward checking removes the value Red of NT and of SA
Forward Checking in Map Coloring

Empty set: the current assignment \{{(WA \leftarrow R), (Q \leftarrow G), (V \leftarrow B)}\}
does not lead to a solution

<table>
<thead>
<tr>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
</tr>
<tr>
<td>R</td>
<td>GB</td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
<td>GB</td>
<td>RGB</td>
</tr>
<tr>
<td>R</td>
<td>B</td>
<td>G</td>
<td>RB</td>
<td>RGB</td>
<td>B</td>
<td>RGB</td>
</tr>
<tr>
<td>R</td>
<td>B</td>
<td>G</td>
<td>RB</td>
<td>B</td>
<td>RGB</td>
<td>RGB</td>
</tr>
</tbody>
</table>
Forward Checking (General Form)

Whenever a pair \((X \leftarrow v)\) is added to assignment \(A\) do:

For each variable \(Y\) not in \(A\) do:

For every constraint \(C\) relating \(Y\) to the variables in \(A\) do:

Remove all values from \(Y\)’s domain that do not satisfy \(C\)

Modified Backtracking Algorithm

CSP-BACKTRACKING(\(A, \text{var-domains}\))

1. If assignment \(A\) is complete then return \(A\)
2. \(X \leftarrow\) select a variable not in \(A\)
3. \(D \leftarrow\) select an ordering on the domain of \(X\)
4. For each value \(v\) in \(D\) do
   a. Add \((X \leftarrow v)\) to \(A\)
   b. \(\text{var-domains} \leftarrow\) forward checking(\(\text{var-domains}, X, v, A\))
   c. If no variable has an empty domain then
      (i) result \(\leftarrow\) CSP-BACKTRACKING(\(A, \text{var-domains}\))
      (ii) If result \(\neq\) failure then return result
   d. Remove \((X \leftarrow v)\) from \(A\)
5. Return failure
Modified Backtracking Algorithm

CSP-BACKTRACKING(A, var-domains)
1. If assignment A is complete then return A
2. X \leftarrow select a variable not in A
3. D \leftarrow select an ordering on the domain of X
4. For each value v in D do
   a. Add (X\leftarrow v) to A
   b. var-domains \leftarrow forward checking(var-domains, X, v, A)
   c. If no variable has an empty domain then
      (i) result \leftarrow CSP-BACKTRACKING(A, var-domains)
      (ii) If result \neq failure then return result
   d. Remove (X\leftarrow v) from A
5. Return failure
Modified Backtracking Algorithm

CSP-BACKTRACKING(A, var-domains)
1. If assignment A is complete then return A
2. X ← select a variable not in A
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X<v) to A
   b. var-domains ← forward checking(var-domains, X, v, A)
   c. If no variable has an empty domain then
      (i) result ← CSP-BACKTRACKING(A, var-domains)
      (ii) If result ≠ failure then return result
   d. Remove (X<v) from A
5. Return failure

1) Which variable \(X_i\) should be assigned a value next?
   → Most-constrained-variable heuristic
   → Most-constraining-variable heuristic

2) In which order should its values be assigned?
   → Least-constraining-value heuristic

NOTE: Different use of the word “heuristic” from A*
Don’t confuse these two! You will only get questions about heuristics as functions from states to reals!
Most-Constrained-Variable Heuristic

1) Which variable $X_i$ should be assigned a value next?

Select the variable with the smallest remaining domain

[Rationale: Minimize the branching factor]

Map Coloring

- SA’s remaining domain has size 1 (value B remaining)
- Q’s remaining domain has size 2
- NSW’s, V’s, and T’s remaining domains have size 3

→ Select SA
Most-Constraining-Variable Heuristic

1) Which variable $X_i$ should be assigned a value next?

Among the variables with the smallest remaining domains (ties with respect to the most-constrained-variable heuristic), select the one that appears in the largest number of constraints on variables not in the current assignment

[Rationale: Increase future elimination of values, to reduce future branching factors]

---

Map Coloring

- Before any value has been assigned, all variables have a domain of size 3, but SA is involved in more constraints (5) than any other variable

→ Select SA and assign a value to it (e.g., Blue)
Least-Constraining-Value Heuristic

2) In which order should X’s values be assigned?

Select the value of X that removes the smallest number of values from the domains of those variables which are not in the current assignment

[Rationale: Since only one value will eventually be assigned to X, pick the least-constraining value first, since it is the most likely not to lead to an invalid assignment]

[Note: Using this heuristic requires performing a forward-checking step for every value, not just for the selected value]

Map Coloring

- Q's domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 values for SA, while assigning Red would leave 1 value
Map Coloring

- Q’s domain has two remaining values: Blue and Red
- Assigning Blue to Q would leave 0 value for SA, while assigning Red would leave 1 value
  \[\Rightarrow\] So, assign Red to Q

More Advanced Constraint Propagation

- Forward checking can’t discover all possible consequences that could lead to failure
  - (Doing this in general would require solving the entire problem, so we shouldn’t expect a free lunch here.)
- AC3 (see textbook) is an advanced algorithm that is a good trade off between efficiency and effectiveness
  - But how hard are CSPs, really?
Digression: NP-Hardness

• NP hardness is not an AI topic
• You will not be tested on it explicitly, but

• It’s important for all computer scientists
• Understanding it will deepen your understanding of AI (and other CS) topics
• You will be expected to understand its relevance and use for AI problems

• Eat your vegetables; they’re good for you

P and NP

• P and NP are about decision problems
• P is set of problems that can be solved in polynomial time
• NP is a superset of P
• NP is the set of problems that:
  – Have solutions which can be verified in polynomial time or, equivalently,
  – can be solved by a non-deterministic Turing machine in polynomial time (OK if you don’t know what that means yet)
• Roughly speaking:
  – Problems in P are tractable – can be solved in a reasonable amount of time, and Moore’s law helps
  – Some problems in NP might not be tractable
Isn’t P big?

- P includes $O(n)$, $O(n^2)$, $O(n^{10})$, $O(n^{100})$, etc.
- Clearly $O(n^{10})$ isn’t something to be excited about – not practical

- Computer scientists are very clever at making things that are in P efficient

- First algorithms for some problems are often quite expensive, e.g., $O(n^3)$, but research often brings this down
Understanding the class NP

• A class of decision problems (Yes/No)
• Solutions can be verified in polynomial time
• Examples:
  – Graph coloring:

  – Sortedness: [1 2 3 4 5 8 7]

NP-hardness

• Many problems in AI are NP-hard (or worse)
• What does this mean?
• NP-hard = as hard as hardest problems in NP
• Identifying a problem as NP hard means:
  – You probably shouldn’t waste time trying to find a polynomial time solution
  – If you find a polynomial time solution, either
    • You have a bug
    • Find a place on your shelf for your Turing award
• NP hardness is a major triumph (and failure) for computer science theory
NP-hardness

• Why it is a failure:
  – There is a huge class of problems with no known efficient solutions
  – We have failed, as a community, to either find efficient solutions or prove that none exist

• Why it is a triumph:
  – We have developed a precise language for talking about these problems
  – We have developed sophisticated ways to reason about and categorize the problems we don’t know how to solve efficiently

P=NP?

• Biggest open question in CS

• Can NP-hard problems be solved in poly time?
  • Probably not, but nobody has been able to prove it yet

• Many false starts, e.g.:
How challenging is “P=NP?”

• Princeton University CS department

Hardness of CSPs

• CSPs are known to be NP-hard (for most reasonable formulations of the problem)
• Bad news: Don’t bother trying to find a general, efficient way to solve CSPs
• Good news: Many problems can be solved much faster than the worst (exponential) case in practice
• So-so news: Sometimes you just need to run a solver and see what happens
  – You might get an answer quickly
  – You might just wait, and wait, and wait...
CSP Conclusions

• CSPs are a general language for describe a large family of problems
• Might require exponential time (worst case)

• Advanced algorithms exist that try to discover bad choices quickly, reducing the search space
  – Microsoft Solver Foundation
  – CPLEX