General Game Playing (GGP)
Winter term 2013/2014

5. Search algorithms I
Outline

<table>
<thead>
<tr>
<th>Date</th>
<th>What will we do?</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.10.2013</td>
<td>Introduction, Repetition propositional logic and FOL</td>
</tr>
<tr>
<td>29.10.2013</td>
<td>Repetition FOL / Datalog and Prolog</td>
</tr>
<tr>
<td>05.11.2013</td>
<td>Game Description Language</td>
</tr>
<tr>
<td>12.11.2013</td>
<td>Design of GDL games</td>
</tr>
<tr>
<td>19.11.2013</td>
<td>Search Algorithms 1</td>
</tr>
<tr>
<td>26.11.2013</td>
<td>No lecture</td>
</tr>
<tr>
<td>03.12.2013</td>
<td>Search Algorithms 2</td>
</tr>
<tr>
<td>10.12.2013</td>
<td>Fluent Calculus and Fluxplayer</td>
</tr>
<tr>
<td>17.12.2013</td>
<td><strong>Midterm competition</strong></td>
</tr>
<tr>
<td>14.01.2014</td>
<td>Meta-Gaming</td>
</tr>
<tr>
<td>21.01.2014</td>
<td>Game Theory</td>
</tr>
<tr>
<td>28.01.2014</td>
<td>?</td>
</tr>
<tr>
<td>04.02.2014</td>
<td><strong>Final Competition</strong></td>
</tr>
<tr>
<td>11.02.2014</td>
<td><strong>Exam</strong></td>
</tr>
</tbody>
</table>
Final decision:

Which part of existing GGP code you would like to use in your own code:

1) Any code

2) Only code up to legal moves (no strategies/heuristics how to determine the next move along legal moves)

3) No existing code at all

Final decision, please!
Search problem formulation

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

- What is the optimal solution?
- What is the state space?
Review: Tree search

- Initialize the **fringe** using the **starting state**
- While the fringe is not empty
  - Choose a fringe node to expand according to **search strategy**
  - If the node contains the **goal state**, return solution
  - Else **expand** the node and add its children to the fringe
Search strategies

• **A search strategy** is defined by picking the order of node expansion

• Strategies are evaluated along the following dimensions:
  – **Completeness**: does it always find a solution if one exists?
  – **Optimality**: does it always find a least-cost solution?
  – **Time complexity**: number of nodes generated
  – **Space complexity**: maximum number of nodes in memory

• Time and space complexity are measured in terms of
  – \( b \): maximum branching factor of the search tree
  – \( d \): depth of the optimal solution
  – \( m \): maximum length of any path in the state space (may be infinite)
Uninformed search
Uninformed search strategies

- **Uninformed** search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening search
Properties of breadth-first search

• **Complete?**
  Yes (if branching factor $b$ is finite)

• **Optimal?**
  Yes – if cost = 1 per step

• **Time?**
  Number of nodes in a $b$-ary tree of depth $d$: $O(b^d)$
  ($d$ is the depth of the optimal solution)

• **Space?**
  $O(b^d)$

• Space is the bigger problem (more than time)
Properties of depth-first search

- **Complete?**
  Fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  \[\rightarrow\text{complete in finite spaces}\]

- **Optimal?**
  No – returns the first solution it finds

- **Time?**
  Could be the time to reach a solution at maximum depth \(m: O(b^m)\)
  Terrible if \(m\) is much larger than \(d\)
  But if there are lots of solutions, may be much faster than BFS

- **Space?**
  \(O(bm)\), i.e., linear space!
**Uniform-cost search**

- Expand least-cost unexpanded node
- Implementation: *fringe* is a queue ordered by path cost (priority queue)
- Equivalent to breadth-first if step costs all equal

**Complete?**
Yes, if step cost is greater than some positive constant $\varepsilon$

**Optimal?**
Yes – nodes expanded in increasing order of path cost

**Time?**
Number of nodes with path cost $\leq$ cost of optimal solution ($C^*$), $O(b^{C^*/\varepsilon})$
This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps

**Space?**
$O(b^{C^*/\varepsilon})$
Iterative deepening search

- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…
Iterative deepening search

Limit = 0
Iterative deepening search
Iterative deepening search

Limit = 2

Diagram showing the iterative deepening search process with a limit of 2.
Iterative deepening search

Limit = 3
Properties of iterative deepening search

• **Complete?**
  Yes

• **Optimal?**
  Yes, if step cost = 1

• **Time?**
  \((d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)\)

• **Space?**
  \(O(bd)\)
Informed search
Informed search

• Idea: give the algorithm “hints” about the desirability of different states
  – Use an evaluation function to rank nodes and select the most promising one for expansion

• Greedy best-first search
• A* search
Heuristic function

- **Heuristic function** $h(n)$ estimates the cost of reaching goal from node $n$
- Example:
Heuristic for the Romania problem
Greedy best-first search

- Expand the node that has the lowest value of the heuristic function $h(n)$
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Properties of greedy best-first search

- **Complete?**
  No – can get stuck in loops
Properties of greedy best-first search

- **Complete?**
  No – can get stuck in loops
- **Optimal?**
  No
Properties of greedy best-first search

• **Complete?**
  No – can get stuck in loops

• **Optimal?**
  No

• **Time?**
  Worst case: $O(b^m)$
  Best case: $O(bd)$ – If $h(n)$ is 100% accurate

• **Space?**
  Worst case: $O(b^m)$
How can we fix the greedy problem?
A* search

• Idea: avoid expanding paths that are already expensive
• The evaluation function $f(n)$ is the estimated total cost of the path through node $n$ to the goal:

$$f(n) = g(n) + h(n)$$

$g(n)$: cost so far to reach $n$ (path cost)
$h(n)$: estimated cost from $n$ to goal (heuristic)
A* search example
A* search example
A* search example
A* search example
A* search example
A* search example
Admissible heuristics

- A heuristic $h(n)$ is **admissible** if for every node $n$, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: straight line distance never overestimates the actual road distance
- Theorem: If $h(n)$ is admissible, $A^*$ is optimal
Properties of A*

- **Complete?**
  Yes – unless there are infinitely many nodes with $f(n) \leq C^*$

- **Optimal?**
  Yes

- **Time?**
  Number of nodes for which $f(n) \leq C^*$ (exponential)

- **Space?**
  Exponential
## Comparison of search strategies

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complete?</th>
<th>Optimal?</th>
<th>Time complexity</th>
<th>Space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>Yes</td>
<td>If all step costs are equal</td>
<td>O(b^d)</td>
<td>O(b^d)</td>
</tr>
<tr>
<td>UCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Number of nodes with g(n) ≤ C*</td>
<td></td>
</tr>
<tr>
<td>DFS</td>
<td>No</td>
<td>No</td>
<td>O(b^m)</td>
<td>O(bm)</td>
</tr>
<tr>
<td>IDS</td>
<td>Yes</td>
<td>If all step costs are equal</td>
<td>O(b^d)</td>
<td>O(bd)</td>
</tr>
<tr>
<td>Greedy</td>
<td>No</td>
<td>No</td>
<td>Worst case: O(b^m) Best case: O(bd)</td>
<td></td>
</tr>
<tr>
<td>A*</td>
<td>Yes</td>
<td>Yes</td>
<td>Number of nodes with g(n)+h(n) ≤ C*</td>
<td></td>
</tr>
</tbody>
</table>
Compare yourself!

- 8/15 puzzle solved with BFS, A*, IDA*, and …
  [Link](http://n-puzzle-solver.appspot.com/)

Keep in mind (assuming branching factor is roughly 3):

<table>
<thead>
<tr>
<th></th>
<th>n = 8</th>
<th>n = 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average solution size:</td>
<td>22</td>
<td>53</td>
</tr>
<tr>
<td>Tree: # states</td>
<td>$3^{22} = 31,381,059,609$</td>
<td>$3^{53} = 147,808,829,414,346,000,000,000,000,000,000$</td>
</tr>
<tr>
<td></td>
<td>$\approx 10^{10}$</td>
<td>$\approx 10^{38}$</td>
</tr>
<tr>
<td>Graph: # unique states</td>
<td>9!/2 = 181,440</td>
<td>16!/2 = 10,461,394,944,000</td>
</tr>
<tr>
<td></td>
<td>$\approx 10^5$</td>
<td>$\approx 10^{13}$</td>
</tr>
</tbody>
</table>
Side note: what is the exact branching factor?

- 15 puzzle has 4 center cells \((b=3)\), 4 corner cells \((b=1)\), and 8 side cells \((b=2)\).
- Therefore, \(b = \frac{4 \times 3 + 4 \times 1 + 8 \times 2}{16} = 2\).
- Is that correct?
Side note: what is the exact branching factor?

• 15 puzzle has 4 center cells \((b=3)\), 4 corner cells \((b=1)\), and 8 side cells \((b=2)\).
• Therefore, \(b = (4*3 + 4*1 + 8*2)/16 = 2\).
• Assumes all blank positions equally likely!
The Correct Answer

- The asymptotic branching factor of the Fifteen Puzzle is 2.13040
- See here for a proof:

Relaxed heuristics
Relaxed Heuristic

- Relaxed problem:
  A problem with fewer restrictions on the actions is called a relaxed problem.

This is similar to the straight-line heuristics in A*, but now we will look at it from a logical point of view.
Systematic Relaxation

• Precondition List
  – A conjunction of predicates that must hold true before the action can be applied

• Add List
  – A list of predicates that are to be added to the description of the world-state as a result of applying the action

• Delete List
  – A list of predicates that are no longer true once the action is applied and should, therefore, be deleted from the state description

• Primitive Predicates
  – ON(x, y) : tile x is on cell y
  – CLEAR(y) : cell y is clear of tiles
  – ADJ(y, z) : cell y is adjacent to cell z
Two simple relaxed models of Sliding Puzzle problems

We can generate two simple relaxed models by removing certain conditions:

Move(x, y, z):

- precondition list: ON(x, y), CLEAR(z), ADJ(y, z)
- add list: ON(x, z), CLEAR(y)
- delete list: ON(x, y), CLEAR(z)
(1) By removing CLEAR(z) and ADJ(y, z), we can derive “Misplaced distance”.

\[
\begin{array}{cccc}
15 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
13 & 14 & 12 \\
\end{array}
\rightarrow
\begin{array}{cccc}
& 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]

Misplaced distance is 1+1=2 moves
(2) By removing CLEAR(z), we can derive “Manhattan distance”.

Manhattan distance is 6+3=9 moves
Three more relaxed heuristic functions

- Pattern Database Heuristics
- Linear Conflict Heuristics
- Gaschnig’s Heuristics
Pattern Database Heuristics

- The idea behind pattern database heuristics is to store the exact solution costs for every possible sub-problem instance.
Example: 8-puzzle

1 2
3 4 5
6 7 8

181,440 states

Domain = blank  1  2  3  4  5  6  7  8
“Patterns”
created by domain mapping

This mapping produces 9 patterns
Pattern Database

Distance to goal

Pattern

Pattern

Distance to goal
Calculating $h(s)$

Given a state in the original problem

Compute the corresponding pattern

and look up the abstract distance-to-goal

Heuristics defined by PDBs are **consistent**, not just admissible.
Abstract Space
Efficiency

Time for the preprocessing to create a PDB is usually negligible compared to the time to solve one problem-instance with no heuristic.

Memory is the limiting factor.
“Pattern” = leave some tiles unique

Domain = blank 1 2 3 4 5 6 7 8
Abstract = blank ■ ■ ■ ■ 6 7 8

How many patterns?
“Pattern” = leave some tiles unique

Domain = blank 1 2 3 4 5 6 7 8
Abstract = blank 6 7 8

3024 patterns
Domain Abstraction

Domain = blank  1  2  3  4  5  6  7  8
Abstract = blank                     6  7  8

30,240 patterns
8-puzzle PDB sizes
(with the blank left unique)

| 9   | 72  | 15120 |
| 252 | 504 | 22680 |
| 630 | 1512| 30240 |
| 1512|     | 45360 |
| 2520|     | 60480 |
| 3024|     | 90720 |
| 3780|     | 181440|
| 5040|     |       |
| 7560|     |       |
| 10080|    |       |
| 15120|    |       |
Automatic Creation of Domain Abstractions

- Easy to enumerate all possible domain abstractions

- They form a lattice, e.g.

  \[
  \begin{array}{cccccccc}
  \text{Domain} = \text{blank} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \text{Abstract} = \text{blank} &  & & & & & & \\
  \end{array}
  \]

  is “more abstract” than the domain abstraction above
Problem: Non-surjectivity

Domain = blank 1 2
Abstract = blank 1 blank
Problem: Non-surjectivity

Domain = blank 1 2
Abstract = blank 1 blank
Problem: Non-surjectivity

Domain = blank  1  2
Abstract = blank  1  blank
Problem: Non-surjectivity

Domain = blank 1 2
Abstract = blank 1 blank
Problem: Non-surjectivity

• Remember: The abstract state space might be as complicated as the original state space
• It is not efficiently decidable whether an abstract state space is surjective
Combining two pattern databases

- Can we just sum up their scores?
- What other alternative do we have?
Disjoint Pattern Database Heuristics

- Two or more patterns that have no tiles in common.

- Add together the heuristic values from the different databases.

- The sum of different heuristics results still be an admissible functions which is closed to the actual optimal cost.
Examples for Disjoint Pattern Database Heuristics

20 moves needed to solve red tiles
25 moves needed to solve blue tiles
Overall heuristic is sum, or $20 + 25 = 45$ moves
A trivial example of disjoint pattern database heuristics is Manhattan Distance in the case that you view every slide as a single pattern database.

Overall heuristic is sum of the Manhattan Distance of each tile which is 39 moves.
A* vs PDB-size

# nodes expanded (A*) vs pattern database size (# of abstract states)
Linear Conflict Heuristic Function

Def. Linear Conflict Heuristic

Two tiles $t_j$ and $t_k$ are in a linear conflict if $t_j$ and $t_k$ are the same line, the goal positions of $t_j$ and $t_k$ are both in that line, $t_j$ is to the right of $t_k$, and goal position of $t_j$ is to the left of the goal position of $t_k$. 
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Linear Conflict Example

Manhattan distance is \(2 + 2 = 4\) moves
Linear Conflict Example

Manhattan distance is $2+2=4$ moves
Consistency of Linear Conflict

The Linear Conflict heuristic is monotone (and therefore admissible).

To establish monotonicity we must show that \( \forall s, s'[f(s') \geq f(s) \) (where \( s' \) is a successor of \( s \)). Recall that \( f(s) = g(s) + h'(s) \), where \( g(s') = g(s) + 1 \) and \( h'(s) = MD(s) + LC(s) \).

In the movement from a state to its successor, let us assume that tile \( x \) moves from row \( r_{old} \) to \( r_{new} \), while remaining in column \( c_k \). There are three cases:

1. The goal position of \( x \) is in neither \( r_{old} \) nor \( r_{new} \).
   \[ md(s', x) = md(s, x) \pm 1. \] LC does not change. Therefore, \( h(s') = h(s) \pm 2 \), and \( f(s') = f(s) + 1 \pm 1 \geq f(s) \).

2. The goal position of \( x \) is in \( r_{new} \).
   \[ md(s', x) = md(s, x) \pm 1. \] Because \( r_{old} \) is not \( x \)'s goal row, its absence has no effect: \( lc(s', r_{old}) = lc(s, r_{old}) \). Because \( x \) is moving into its goal row, either \( lc(s', r_{new}) = lc(s, r_{new}) \) or \( lc(s', r_{new}) = lc(s, r_{new}) + 2 \). Therefore, \( h(s') = h(s) \pm 1 \), and \( f(s') = f(s) + 1 \pm 1 \geq f(s) \).

3. The goal position of \( x \) is in \( r_i \).
   \[ md(s', x) = md(s, x) + 1. \] Because \( r_{old} \) is \( x \)'s goal row, either \( lc(s', r_{old}) = lc(s, r_{old}) \) or \( lc(s', r_{old}) = lc(s, r_{old}) - 2 \). Because \( r_{new} \) is not \( x \)'s goal row, \( lc(s', r_{new}) = lc(s, r_{new}) \). Therefore, \( h(s') = h(s) \pm 1 \), and \( f(s') = f(s) + 1 \pm 1 \geq f(s) \).

In all cases, \( f(s') \geq f(s) \). Similarly for movement within a row.

Q.E.D.
Linear Conflict Heuristic Function

- The linear conflict heuristic will cost at least 2 more than Manhattan distance.

- Linear conflict heuristic is more accurate or more informative than just using Manhattan Distance since it is closer to the actual optimal cost.
Gaschnig’s Heuristic Function

- Gaschnig introduced the 9MAXSWAP problem.
- The relaxed problem assumes that a tile can move from square A to B if B is blank, but A and B do not need to be adjacent.
- It underestimates the distance function of 8-puzzle, it is a closer approximation of the 8-puzzle’s distance.
Maxsort Algorithm

One algorithm to solve 9MAXSWAP is Maxsort.

- P: the current permutation
- B: the location of element i in the permutation Array.
Apply MAXSORT as A Heuristic

To apply MAXSORT as a heuristic for the 8-puzzle, we take the number of switches as the heuristic cost at any search node.
Gaschnig Heuristic Example

Current Node: 29613478

Goal Node: 123456789
Gaschnig Heuristic Function

- In the previous example, the Gaschnig Heuristic cost for the node on the right side is 7 which is just the number of switches to make the sequence 296134758 to be 123456789. (9 means blank)
# Comparison of Heuristic Estimates

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>Misplaced Tiles</th>
<th>Relaxed Adjacency (Gaschnig’s)</th>
<th>Manhattan Distance (Subset of Pattern)</th>
<th>Linear Conflict</th>
<th>Actual Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 1</td>
<td>2 1</td>
<td>4 3 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 4 5</td>
<td>5 4 3</td>
<td>8 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 3 8</td>
<td>6 7 8</td>
<td>5 2 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>5.75</td>
<td>4.25</td>
<td>5.33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Misplaced Tiles**: A measure of how many tiles are out of place.
- **Relaxed Adjacency (Gaschnig’s)**: A heuristic estimate that considers the adjacency of tiles.
- **Manhattan Distance (Subset of Pattern)**: A heuristic estimate based on the Manhattan distance.
- **Linear Conflict**: A heuristic estimate based on the linear conflict of tiles.
- **Actual Distance**: The actual number of moves required to solve the puzzle.

The table above compares different heuristic estimates for solving the tile puzzle game, showing the average number of misplaced tiles and the variance ($\sigma^2$) for each heuristic.
# Comparison of Heuristic Estimates

<table>
<thead>
<tr>
<th>Puzzle</th>
<th>2</th>
<th>4</th>
<th>1</th>
<th>5</th>
<th>8</th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misplaced Tiles</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relaxed Adjacency (Gaschnig’s)</td>
<td>6</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manhattan Distance (Subset of Pattern)</td>
<td>6</td>
<td>6</td>
<td>22</td>
<td>14</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear Conflict</td>
<td>8</td>
<td>12</td>
<td>22</td>
<td>24</td>
<td>16.5</td>
<td>59.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual Distance</td>
<td>22</td>
<td>20</td>
<td>26</td>
<td>26</td>
<td>23.5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average</th>
<th>$\sigma^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.25</td>
<td>5.33</td>
</tr>
<tr>
<td>5.75</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>16.5</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgements

- Svetlana Lazebnik: Solving problems by searching (COMP 590 Artificial Intelligence)
- Shaun Gause, Yu Cao: Heuristics for sliding-tile puzzles (University of South Carolina)
- Richard Korf: Recent Progress in the Design and Analysis of Admissible Heuristic Functions