py_search Documentation

Release 1.2.0

Christopher J. MacLellan

Oct 09, 2017

Contents

1 Installation **3**

2 Important Links **5**

2.1 Py-Search Package 5

2.2 Test Problems 13

2.3 Examples 17

3 Indices and tables **25**

Python Module Index **27**

This is a Python library of algorithms that implements various search algorithms written by Christopher MacLellan (http://www.christopia.net). In particular, there are uninformed, informed, and optimization techniques implemented with a standard interface.

CHAPTER 1

Installation

You can install this software using pip:

```
pip install -U py_search
```

You can install the latest version of the code directly from github:

```
pip install -U git+https://github.com/cmaclell/py_search@master
```

CHAPTER 2

Important Links

- Source code: https://github.com/cmaclell/py_search
- Documentation: http://py_search.readthedocs.org

Contents:

Py-Search Package

Base

This module contains the data_structures used in py_search. In particular, it contains the the *Problem* class, which is used to represent the different search problems, and the *AnnotatedProblem* class, which wraps around a specific problem and keeps track of the number of core method calls.

At a lower level this module also contains the *Node* class, which is used to represent a node in the search space.

Finally, the module contains the *Fringe* class, and its instantiations (*FIFOQueue*, *LIFOQueue*, and PrioritySet). A Fringe is used to structure the way a search space is explored.

class py_search.base.**AnnotatedProblem**(*problem*)

Bases: *py_search.base.Problem*

A Problem class that wraps around another Problem and keeps stats on nodes expanded and goal tests performed.

goal_test(*node*)

A wrapper for the goal_test method that keeps track of the number of goal_tests performed.

node_value(*node*)

A wraper for the node value method that keeps track of the number of times a node value was calculated.

random_node()

A wrapper for the random_node method.

random_successor(*node*)
A wrapper for the random_successor method that keeps track of the number of nodes expanded.

successors(*node*)
A wrapper for the successor method that keeps track of the number of nodes expanded.

class py_search.base.**FIFOQueue**
Bases: *py_search.base.Fringe*

A first-in-first-out queue. Used to get breadth first search behavior.

```
>>> fifo = FIFOQueue()
>>> fifo.push(0)
>>> fifo.push(1)
>>> fifo.push(2)
>>> print(fifo.pop())
0
>>> print(fifo.pop())
1
>>> print(fifo.pop())
2
```

pop()

push(*node*)

remove(*node*)

class py_search.base.**Fringe**
Bases: object

A template for a fringe class. Used to control the strategy of different search approaches.

extend(*nodes*)
Given an iterator (*nodes*) adds all the nodes to the collection.

pop()
Pops a node off the collection.

push(*node*)
Adds one node to the collection.

class py_search.base.**LIFOQueue**
Bases: *py_search.base.FIFOQueue*

A last-in-first-out queue. Used to get depth first search behavior.

```
>>> lifo = LIFOQueue()
>>> lifo.push(0)
>>> lifo.push(1)
>>> lifo.push(2)
>>> print(lifo.pop())
2
>>> print(lifo.pop())
1
>>> print(lifo.pop())
0
```

pop()

class py_search.base.**Node**(*state*, *parent=None*, *action=None*, *node_cost=0*, *extra=None*)
Bases: object

A class to represent a node in the search. This node stores state information, path to the state, cost of the node, depth of the node, and any extra information.

Parameters

- **state** (*object for tree search and hashable object for graph search*) – the state at this node
- **parent** (*Node*) – the node from which the current node was generated
- **action** (*typically a string, but can be any object*) – the action performed to transition from parent to current.
- **cost** (*float*) – the cost of reaching the current node
- **extra** (*object*) – extra information to store in this node, typically used to store non-hashable information about the state.

cost()
Returns the cost of the current node.

depth()
Returns the depth of the current node. Uses a loop to compute depth (no tail recursion in python).

path()
Returns a path (tuple of actions) from the initial to current node.

class py_search.base.**PriorityQueue**(*node_value=<function <lambda>>, cost_limit=inf, max_length=inf*)
Bases: *py_search.base.Fringe*

A priority queue that sorts elements by their value. Always returns the minimum value item. A *PriorityQueue* accepts a node_value function, a cost_limit (nodes with a value greater than this limit will not be added) and a max_length parameter. If adding an item ever causes the size to exceed the max_length then the worst nodes are removed until the list is equal to max_length.

```
>>> pq = PriorityQueue(node_value=lambda x: x, max_length=3)
>>> pq.push(6)
>>> pq.push(0)
>>> pq.push(2)
>>> pq.push(6)
>>> pq.push(7)
>>> print(len(pq))
3
>>> print(pq.pop())
0
>>> print(pq.pop())
2
>>> print(pq.pop())
6
```

Parameters

- **node_value** (*a function with one parameter for node*) – The node evaluation function (defaults to `lambda x: x.cost()`)
- **cost_limit** (*float*) – the maximum value for elements in the set, if an item exceeds this limit then it will not be added (defaults to `float('inf')`)
- **max_length** (int or `float('inf')`) – The maximum length of the list (defaults to `float('inf')`

clear()
Empties the list.

peek()
Returns the best node.

peek_value()
Returns the value of the best node.

pop()
Pop the best value from the priority queue.

push(*node*)
Push a node into the priority queue. If the node exceeds the cost limit then it is not added. If the max_length is exceeded by adding the node, then the worst node is discarded from the set.

update_cost_limit(*cost_limit*)
Updates the cost limit and removes any nodes that violate the new limit.

class py_search.base.**Problem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: object

The basic problem to solve. The main functions that must be defined include successors and goal_test. Some search techniques also require the random_successor and predecessors methods to be implemented.

goal_test(*node*)
Returns true if a goal state is found. This is typically used not used by the local search / optimization techniques.

node_value(*node*)
Returns the the value of the current node. This is the value being minimized by the search. By default the cost is used, but this function can be overloaded to include a heuristic.

random_node()
This method returns a random node in the search space. This is used by some of the local search / optimization techniques.

random_successor(*node*)
This method should return a single successor node. This is used by some of the local search / optimization techniques.

successors(*node*)
An iterator that yields all of the successors of the current node.

Uninformed Search

This module includes the core search methods *tree_search()* and *graph_search* and the primary uninformed search techniques: *depth_first_search()*, *breadth_first_search()*, and *iterative_deepening_search()*.

py_search.uninformed.**breadth_first_search**(*problem*, *depth_limit=inf*, *search=<function graph_search>*)
A simple implementation of depth-first search using a FIFO queue.

Parameters

- **problem** (Problem) – The problem to solve.
- **search** (*graph_search()* or :func‘tree_search‘) – A search algorithm to use (defaults to graph_search).

- **depth_limit** (*int or float('inf')*) – A limit for the depth of the search tree. If set to float('inf'), then depth is unlimited.

py_search.uninformed.**depth_first_search**(*problem*, *depth_limit=inf*, *search=<function graph_search>*)

A simple implementation of depth-first search using a LIFO queue.

Parameters

- **problem** (Problem) – The problem to solve.
- **search** (*graph_search()* or :func'tree_search') – A search algorithm to use (defaults to graph_search).
- **depth_limit** (*int or float('inf')*) – A limit for the depth of the search tree. If set to float('inf'), then depth is unlimited.

py_search.uninformed.**graph_search**(*problem*, *fringe*, *depth_limit=inf*)

Perform graph search (i.e., no duplicate states) using the given fringe class. Returns an iterator to the solutions, so more than one solution can be found.

Note that the closed list will allow re-expansion of nodes with a lower cost.

Parameters

- **problem** (Problem) – The problem to solve.
- **fringe** (fringe) – The fringe class to use.
- **depth_limit** (*int or float('inf')*) – A limit for the depth of the search tree. If set to float('inf'), then depth is unlimited.

py_search.uninformed.**iterative_deepening_search**(*problem*, *search=<function graph_search>*, *initial_depth_limit=0*, *depth_inc=1*, *max_depth_limit=inf*)

An implementation of iterative deepening search. This search is basically depth-limited depth first up to the depth limit. If no solution is found at the current depth limit then the depth limit is increased by depth_inc and the depth-limited depth first search is restarted.

Parameters

- **problem** (Problem) – The problem to solve.
- **search** (*graph_search()* or :func'tree_search') – A search algorithm to use (defaults to graph_search).
- **initial_depth_limit** (*int or float('inf')*) – The initial depth limit for the search.
- **depth_inc** (*int*) – The amount to increase the depth limit after failure.
- **max_depth_limit** (*int or float('inf')*) – The maximum depth limit (default value of *float('inf')*)

py_search.uninformed.**tree_search**(*problem*, *fringe*, *depth_limit=inf*)

Perform tree search (i.e., search where states might be duplicated) using the given fringe class.Returns an iterators to the solutions, so more than one solution can be found.

Parameters

- **problem** (Problem) – The problem to solve.
- **fringe** (fringe) – The fringe class to use.
- **depth_limit** (*int or float('inf')*) – A limit for the depth of the search tree. If set to float('inf'), then depth is unlimited.

Informed Search

This module includes the informed search techniques: *best_first_search()* (i.e., A*), *iterative_deepening_best_first_search()* (i.e., IDA*), *beam_search()*, and *widening_beam_search()*.

py_search.informed.**beam_search**(*problem*, *beam_width=1*, *graph_search=True*)

A variant of breadth-first search where all nodes in the fringe are expanded, but the resulting new fringe is limited to have length beam_width, where the nodes with the worst value are dropped. The default beam width is 1, which yields greedy best-first search (i.e., hill climbing).

There are different ways to implement beam search, namely best-first beam search and breadth-first beam search. According to:

> Wilt, C. M., Thayer, J. T., & Ruml, W. (2010). A comparison of greedy search algorithms. In Third Annual Symposium on Combinatorial Search.

breadth-first beam search almost always performs better. They find that allowing the search to re-expand duplicate nodes if they have a lower cost improves search performance. Thus, our implementation is a breadth-first beam search that re-expand duplicate nodes with lower cost.

Parameters

- **problem** (Problem) – The problem to solve.
- **beam_width** (*int*) – The size of the beam (defaults to 1).
- **graph_search** (*boolean*) – whether to use graph or tree search.

py_search.informed.**best_first_search**(*problem*, *search=<function graph_search>*, *cost_limit=inf*)

Cost limited best-first search. By default the cost limit is set to *float('inf')*, so it is identical to traditional best-first search. This implementation uses a priority set (i.e., a sorted list without duplicates) to maintain the fringe.

If the problem.node_value is the cost to reach the node plus an admissible heuristic estimate of the distance from the node to the goal, then this is the A* algorithm.

Parameters

- **problem** (Problem) – The problem to solve.
- **search** (graph_search() or :func'tree_search') – A search algorithm to use (defaults to graph_search).
- **cost_limit** (*float*) – The cost limit for the search (default = *float('inf')*)

py_search.informed.**iterative_deepening_best_first_search**(*problem*, *search=<function graph_search>*, *initial_cost_limit=0*, *cost_inc=1*, *max_cost_limit=inf*)

A variant of iterative deepening that uses cost to determine the limit for expansion. When search fails, the cost limit is increased according to *cost_inc*. If the heuristic is admissible, then this is guranteed to find the best solution (similar to best first serach), but uses less memory.

If the problem.node_value is the cost to reach the node plus an admissible heuristic estimate of the distance from the node to the goal, then this is the IDA* algorithm.

Parameters

- **problem** (Problem) – The problem to solve.

- **search** (graph_search() or :func'tree_search') – A search algorithm to use (defaults to graph_search).
- **initial_cost_limit** (*float*) – The initial cost limit for the search.
- **cost_inc** (*float*) – The amount to increase the cost limit after failure.
- **max_cost_limit** (*float*) – The maximum cost limit (default value of *float('inf')*)

py_search.informed.**widening_beam_search**(*problem*, *initial_beam_width=1*, *max_beam_width=1000*, *graph_search=True*)

A variant of beam search that successively increase the beam width when search fails. This ensures that if a solution exists, and you're using graph search, and the solution space is finite, then that beam search will find it.

However, if you are looking for multiple solutions, then it might return duplicates as the width is increased. As the beam width increase, behavior is closer and closer to breadth-first search.

Parameters

- **problem** (Problem) – The problem to solve.
- **initial_beam_width** (*int*) – The initial size of the beam (defaults to 1).
- **max_beam_width** (*int*) – The maximum size of the beam (defaults to 1000).

Optimization / Local Search

This module contains the local search / optimization techniques. Instead of trying to find a goal state, these algorithms try to find the lowest cost state.

py_search.optimization.**branch_and_bound**(*problem*, *graph_search=True*, *depth_limit=inf*)

An exhaustive optimization technique that is guranteed to give the best solution. In general the algorithm starts with some (potentially non-optimal) solution. Then it uses the cost of the current best solution to prune branches of the search that do not have any chance of being better than this solution (i.e., that have a node_value > current best cost).

In this implementation, node_value should provide an admissible lower bound on the cost of solutions reachable from the provided node. If node_value is inadmissible, then optimality guarantees are lost.

Also, if the search space is infinite and/or the node_value function provides too little guidance (e.g., node_value = float('-inf')), then the search might never terminate. To counter this, a depth_limit can be provided that stops expanding nodes after the provided depth. This will ensure the search is finite and guaranteed to terminate.

Finally, the problem.goal_test function can be used to terminate search early if a good enough solution has been found. If goal_test(node) return True, then search is immediately terminated and the node is returned.

Note, the current implementation uses best-first search via a priority queue data structure.

Parameters

- **problem** (*py_search.base.Problem*) – The problem to solve.
- **graph_search** (*Boolean*) – Whether to use graph search (no duplicates) or tree search (duplicates)

py_search.optimization.**hill_climbing**(*problem*, *random_restarts=0*, *max_sideways=0*, *graph_search=True*)

Probably the simplest optimization approach. It expands the list of neighbors and chooses the best neighbor (steepest descent hill climbing).

Default configuration should yield similar behavior to *local_beam_search()* when it has a width of 1, but doesn't need to maintain alternatives, so might use slightly less memory (just stores the best node instead of limited length priority queue).

If graph_search is true (the default), then a closed list is maintained. This is imporant for search spaces with platues because it keeps the algorithm from reexpanding neighbors with the same value and getting stuck in a loop.

If random_restarts > 0, then search is restarted multiple times. This can be useful for getting out of local minimums.

The problem.goal_test function can be used to terminate search early if a good enough solution has been found. If goal_test(node) return True, then search is immediately terminated and the node is returned.

Parameters

- **problem** (*py_search.base.Problem*) – The problem to solve.
- **random_restarts** (*int*) – The number of times to restart search. The initial state is used for the first search and subsequent starts begin at a random state.
- **max_sideways** (*int*) – Specifies the max number of contiguous sideways moves.
- **graph_search** (*Boolean*) – Whether to use graph search (no duplicates) or tree search (duplicates)

py_search.optimization.**local_beam_search**(*problem*, *beam_width=1*, *max_sideways=0*, *graph_search=True*)

A variant of py_search.informed_search.beam_search() that can be applied to local search problems. When the beam width of 1 this approach yields behavior similar to *hill_climbing()*.

The problem.goal_test function can be used to terminate search early if a good enough solution has been found. If goal_test(node) return True, then search is immediately terminated and the node is returned.

Parameters

- **problem** (*py_search.base.Problem*) – The problem to solve.
- **beam_width** (*int*) – The size of the search beam.
- **max_sideways** (*int*) – Specifies the max number of contiguous sideways moves.
- **graph_search** (*Boolean*) – Whether to use graph search (no duplicates) or tree search (duplicates)

py_search.optimization.**random**() → x in the interval [0, 1).

py_search.optimization.**simulated_annealing**(*problem*, *temp_factor=0.95*, *temp_length=None*, *initial_temp=None*, *init_prob=0.4*, *min_accept=0.02*, *min_change=1e-06*, *limit=inf*)

A more complicated optimization technique. At each iteration a random successor is expanded if it is better than the current node. If the random successor is not better than the current node, then it is expanded with some probability based on the temperature.

Used the formulation of simulated annealing found in: Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1989). Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. Operations research, 37(6), 865-892.

Also, the problem.goal_test function can be used to terminate search early if a good enough solution has been found. If goal_test(node) return True, then search is immediately terminated and the node is returned.

Parameters

- **problem** (*py_search.base.Problem*) – The problem to solve.
- **temp_factor** (*float*) – The factor for geometric cooling, a value between 0 and 1, but usually very close to 1.

- **temp_length** (*int*) – The number of nodes to expand at each temperature. If set to *None* (the default) then it is automatically chosen to be equal to the length of the successors list.
- **initial_temp** (*float or None*) – The initial temperature for the annealing. The number is objective function specific. If set to None (the default), then a semi-random walk is used to select an initial temperature that will yield approx. init_prob acceptance rate for worse states.
- **min_accept** (*float between 0 and 1*) – The fraction of states that must be accepted in temp_length iterations (taken from a single temperature) to not be frozen. Every time this is not exceeded, the frozen counter is incremented until it hits 5. If a better state is found, then the frozen counter is reset to 0.
- **min_change** (*float*) – The amount of change that must be achieved in temp_length iterations (taken from a single temperature) to not be frozen. Everytime this is not exceeded, the frozen counter is incremented until it hits 5. If a better state is found, then the frozen counter is reset to 0.
- **limit** (*float*) – The maximum number of iterations (random neighbors) to expand before stopping.

Test Problems

Eight Puzzle

class py_search.problems.eight_puzzle.**EightPuzzle**

An eight puzzle class that can be used to test different search algorithms. When first created the puzzle is in the solved state.

copy()

Makes a deep copy of an EightPuzzle object.

executeAction(*action*)

Executes an action to the EightPuzzle object.

Parameters **action** (*"up", "left", "right", or "down"*) – the action to execute

legalActions()

Returns an iterator to the legal actions that can be executed in the current state.

randomize(*num_shuffles*)

Randomizes an EightPuzzle by executing a random action *num_suffles* times.

class py_search.problems.eight_puzzle.**EightPuzzleProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)

Bases: *py_search.base.Problem*

This class wraps around an Eight Puzzle object and instantiates the successor and goal test functions necessary for conducting search.

This class also implements an heuristic function which is used to compute the value for each successor as cost to node + heuristic estimate of distance to goal. This yield A* search when used with best first search or a more greedy variant when used with Beam Search.

goal_test(*node*)

Check if the goal state has been reached.

misplaced_tile_heuristic(*state*)
The misplaced tiles heuristic.

node_value(*node*)
The function used to compute the value of a node.

successors(*node*)
Computes successors and computes the value of the node as cost + heuristic, which yields A* search when using best first search.

class py_search.problems.eight_puzzle.**NoHeuristic**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: *py_search.problems.eight_puzzle.EightPuzzleProblem*

A variation on the Eight Puzzle Problem that has a heuristic for 0. This yields something equivelent to dijkstra's algorithm when used with best first search and a more greedy variant when used with Beam Search.

node_value(*node*)

N-Queens Problem

class py_search.problems.nqueens.**LocalnQueensProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: *py_search.base.Problem*

A class that wraps around the nQueens object. This version of the problem starts with an empty board and then progressively adds queens.

goal_test(*node*)
Check if the goal state (i.e., no queen conflicts) has been reached.

random_node()

random_successor(*node*)
Generate all permutations of rows.

successors(*node*)
Generate all permutations of rows.

class py_search.problems.nqueens.**nQueens**(*n*)
An nQueens puzzle object

copy()
Makes a deep copy of an nQueens object.

num_conflicts()
Returns a count of the number of conflicts. First checks if there are column conflicts (row conflicts are impossible because of representation). Then checks for diagonals.

randomize()
Randomizes an nQueens by shuffling a list of row to columns assignments.

class py_search.problems.nqueens.**nQueensProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: *py_search.base.Problem*

A class that wraps around the nQueens object. This version of the problem starts with an empty board and then progressively adds queens.

goal_test(*node*)
Check if the goal state (i.e., no queen conflicts) has been reached.

node_value(*node*)
The function used to compute the value of a node.

successors(*node*)
Generate all possible next queen states.

Assignment Problem

class py_search.problems.assignment_problem.**AssignmentProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)

Bases: *py_search.base.Problem*

A tree search version of the assignment problem. Starts with an initially empty assignment and then incrementally builds the assignment up adding one assignment per expansion.

goal_test(*node*)
A test of whether a complete assignment has been reached.

min_cost_heuristic(*node*)
A huristic specifying the minimum cost that could be achieved for unassigned rows.

node_value(*node*)
The value of a node is the combination of the node cost and the min_cost heuristic

successors(*node*)
An iterator that yields the sucessors of the provided node.

class py_search.problems.assignment_problem.**LocalAssignmentProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)

Bases: *py_search.base.Problem*

This class represents a local search version of the assignment problem. I.e., a random state is generated to start the search and then neighbors of the state can be expanded in order to reduce the solution cost.

goal_test(*node*)

node_value(*node*)
Returns a lower bound on the solution cost reachable from the given node (or its children)

random_node()
Generates a node that has a random assignment.

random_successor(*node*)
A function that returns a random successor of the current node. This is used by the simulated annealing function, so it doesn't have to expand all successors.

A successor is generated by randomly flipping a pair of row to column assignments.

successors(*node*)
Generates successor states by flipping each pair of row to column assignments.

py_search.problems.assignment_problem.**cost**(*assignment*, *costs*)
Given an assignemnt and a cost matrix, returns the cost of the assignment.

py_search.problems.assignment_problem.**print_matrix**(*m*)
Print a matrix

py_search.problems.assignment_problem.**random_assignment**(*n*)
Returns a random valid assignment for an n x n matrix

py_search.problems.assignment_problem.**random_matrix**(*n*)
Generates an a list of list of floats (representing an n x n matrix) where the values have mean 0 and std 1.

This is used as cost matrix for an assignment problem.

Graph Partition Problem

class py_search.problems.graph_partition.**LocalGraphPartitionProblem**(*initial*, *parent=None*, *action=None*, *initial_cost=0*, *extra=None*)
Bases: *py_search.base.Problem*

This class represents a local search version of the graph partition problem. I.e., a random state is generated to start the search and then neighbors of the state can be expanded in order to reduce the solution cost.

goal_test(*node*)
The search should never terminate early.

node_value(*node*)
This function is used by the branch_and_bound approach to determine whether successor nodes have the potential to be better than the current node. If this just returns the node cost, then the algorithm will explore nodes greedily.

random_node()
Generates a node that has a random assignment.

random_successor(*node*)
A function that returns a random successor of the current node. This is used by the simulated annealing function, so it doesn't have to expand all successors.

A successor is generated by randomly flipping a pair of row to column assignments.

successors(*node*)
Generates successor states by flipping each pair of row to column assignments.

py_search.problems.graph_partition.**cutsize**(*E*, *p*)

py_search.problems.graph_partition.**generate_graph**(*n*, *p*)
Generates a random graph for graph partitioning. n specifies the number of nodes and p specifies the probability that any pair of nodes has a transition.

py_search.problems.graph_partition.**random**() → x in the interval [0, 1).

py_search.problems.graph_partition.**random_partition**(*V*)

Examples

Eight Puzzle Search Example

```
In [1]: from py_search.problems.eight_puzzle import EightPuzzle
   ...: from py_search.problems.eight_puzzle import EightPuzzleProblem
   ...: from py_search.utils import compare_searches
   ...: from py_search.uninformed import depth_first_search
   ...: from py_search.uninformed import breadth_first_search
   ...: from py_search.uninformed import iterative_deepening_search
   ...: from py_search.informed import best_first_search
   ...: from py_search.informed import iterative_deepening_best_first_search
   ...: from py_search.informed import widening_beam_search
   ...: puzzle = EightPuzzle()
   ...: puzzle.randomize(20)
   ...: initial = puzzle
   ...: print("Eight puzzle being solved:")
   ...: print(puzzle)
   ...: print()
   ...:
   ...: compare_searches(problems=[EightPuzzleProblem(initial)],
   ...:                  searches=[depth_first_search,
   ...:                            breadth_first_search,
   ...:                            iterative_deepening_search,
   ...:                            best_first_search,
   ...:                            iterative_deepening_best_first_search,
   ...:                            widening_beam_search])
   ...:
Eight puzzle being solved:
245
318
670
()
Problem             Search Alg                                Goal Tests    Nodes␣
↪Expanded    Nodes Evaluated    Solution Cost    Runtime
------------------  ---------------------------------------  ------------  -------------
↪---  -----------------  ---------------  ---------
EightPuzzleProblem  depth_first_search                          220
↪635                   0              202     0.0118
EightPuzzleProblem  breadth_first_search                       1716
↪4596                  0               12     0.1117
EightPuzzleProblem  iterative_deepening_search                 3469
↪5651                  0               12     0.1458
EightPuzzleProblem  best_first_search                            78
↪204                 124               12     0.0059
EightPuzzleProblem  iterative_deepening_best_first_search      262
↪694                 446               12     0.0183
EightPuzzleProblem  widening_beam_search                       475
↪1330                843               46     0.0295
```

N-Queens Search Example

```
In [1]: from py_search.problems.nqueens import nQueens
   ...: from py_search.problems.nqueens import nQueensProblem
   ...: from py_search.problems.nqueens import LocalnQueensProblem
```

```
...: from py_search.uninformed import depth_first_search
...: from py_search.uninformed import breadth_first_search
...: from py_search.informed import best_first_search
...: from py_search.informed import beam_search
...: from py_search.optimization import simulated_annealing
...: from py_search.optimization import hill_climbing
...: from py_search.optimization import branch_and_bound
...: from py_search.optimization import local_beam_search
...: from py_search.utils import compare_searches
...:
...: print("####################")
...: print("BACKTRACKING SEARCH")
...: print("####################")
...: initial = nQueens(5)
...: print("Empty %i-Queens Problem" % initial.n)
...: print(initial)
...: print()
...: compare_searches(problems=[nQueensProblem(initial)],
...:                  searches=[depth_first_search,
...:                            breadth_first_search,
...:                            best_first_search,
...:                            beam_search])
...: print()
...: print("############################")
...: print("LOCAL SEARCH / OPTIMZATION")
...: print("############################")
...: initial = nQueens(10)
...: initial.randomize()
...: cost = initial.num_conflicts()
...: print("Random %i-Queens Problem" % initial.n)
...: print(initial)
...: print()
...:
...: def beam2(problem):
...:     return local_beam_search(problem, beam_width=2)
...:
...: def steepest_hill(problem):
...:     return hill_climbing(problem)
...:
...: def annealing(problem):
...:     size = problem.initial.state.n
...:     n_neighbors = (size * (size-1)) // 2
...:     return simulated_annealing(problem,
...:                                initial_temp=1.8,
...:                                temp_length=n_neighbors)
...:
...: def greedy_annealing(problem):
...:     size = problem.initial.state.n
...:     n_neighbors = (size * (size-1)) // 2
...:     return simulated_annealing(problem,
...:                                initial_temp=0,
...:                                temp_length=n_neighbors)
...:
...: compare_searches(problems=[LocalnQueensProblem(initial,
...:                                                initial_cost=cost)],
...:                  searches=[best_first_search,
...:                            branch_and_bound,
...:                            beam2,
```

```
   ...:                             steepest_hill,
   ...:                             annealing, greedy_annealing])
   ...: print()
   ...: initial = nQueens(20)
   ...: initial.randomize()
   ...: cost = initial.num_conflicts()
   ...: print("Random %i-Queens Problem" % initial.n)
   ...: print(initial)
   ...: print()
   ...: compare_searches(problems=[LocalnQueensProblem(initial,
   ...:                                                initial_cost=cost)],
   ...:                  searches=[steepest_hill,
   ...:                            annealing, greedy_annealing])
   ...:
####################
BACKTRACKING SEARCH
####################
Empty 5-Queens Problem
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

()
Problem         Search Alg              Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated  Solution Cost    Runtime
-------------  --------------------  ------------  ----------------  ----------------
↪-  ---------------  ---------
nQueensProblem  depth_first_search              54               134                ␣
↪0  0                0.0060
nQueensProblem  breadth_first_search          1437              5225                ␣
↪0  0                0.1214
nQueensProblem  best_first_search              790              3658                ␣
↪1400  0                0.1872
nQueensProblem  beam_search                     6                55                ␣
↪56  Failed           Failed
()
#########################
LOCAL SEARCH / OPTIMZATION
#########################
Random 10-Queens Problem
| |Q| | | | | | | | |
|Q| | | | | | | | | |
| | | | | | | | | |Q|
| | |Q| | | | | | | |
| | | |Q| | | | | | |
| | | | | |Q| | | | |
| | | | | | | |Q| | |
| | | | | | |Q| | | |
| | | | | | | | |Q| |
| | | | |Q| | | | | |

()
Problem              Search Alg           Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated    Solution Cost    Runtime
-------------------  -----------------  ------------  ----------------  --------------
↪---  ---------------  ---------
```

```
LocalnQueensProblem  best_first_search             4                135
↪133                 0     0.0095
LocalnQueensProblem  branch_and_bound             4                135
↪133                 0     0.0045
LocalnQueensProblem  beam2                        3                 90
↪  0                 0     0.0052
LocalnQueensProblem  steepest_hill               16                 93
↪  0                 0     0.0026
LocalnQueensProblem  annealing                   23                 28
↪  0                 0     0.001
LocalnQueensProblem  greedy_annealing            18                 27
↪  0                 0     0.001
()
Random 20-Queens Problem
| | | | |Q| | | | | | | | | | | | | | |
|Q| | | | | | | | | | | | | | | | | |
| | | | | |Q| | | | | | | | | | | | | |
| | | | | | | | | | | | |Q| | | | | | |
| | | | | | | | | | | | | | | |Q| | | |
| | | | | | | | | | | | | | | |Q| | | | |
| |Q| | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | |Q| | | | | | |
| | | | | | | | | | | | | | |Q| | | | | |
| | | | | | | | | | | |Q| | | | | | | | |
| | | |Q| | | | | | | | | | | | | | | | |
| | | | | | | | |Q| | | | | | | | | | | |
| | | | | | | | |Q| | | | | | | | | | | |
| | | | | | | | | | | | | | | | | |Q| | |
| | | | | | | | | | | | | | | | | | | |Q|
| | | | | | | | | |Q| | | | | | | | | | |
| | | | | | | | | | | | | | | | | | |Q| |
| | | | | | | | | | |Q| | | | | | | | | |

()
Problem              Search Alg          Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated    Solution Cost    Runtime
-------------------  ----------------  ------------  ----------------  ---------------
↪--  ---------------  ---------
LocalnQueensProblem  steepest_hill              36               422
↪ 0                  0     0.0305
LocalnQueensProblem  annealing                  20                23
↪ 0                  0     0.0018
LocalnQueensProblem  greedy_annealing           11                56
↪ 0                  0     0.0044
```

Assignment Problem Optimization Example

```
In [1]:     from munkres import Munkres
   ...:     from py_search.problems.assignment_problem import random_matrix
   ...:     from py_search.problems.assignment_problem import print_matrix
   ...:     from py_search.problems.assignment_problem import cost
   ...:     from py_search.problems.assignment_problem import random_assignment
   ...:     from py_search.problems.assignment_problem import LocalAssignmentProblem
   ...:     from py_search.problems.assignment_problem import AssignmentProblem
```

```
   ...:     from py_search.optimization import local_beam_search
   ...:     from py_search.optimization import simulated_annealing
   ...:     from py_search.optimization import hill_climbing
   ...:     from py_search.informed import beam_search
   ...:     from py_search.informed import best_first_search
   ...:     from py_search.utils import compare_searches
   ...:
   ...:     n = 8
   ...:     costs = random_matrix(n)
   ...:
   ...:     print("####################################################")
   ...:     print("Optimial solution using Munkres/Hungarian Algorithm")
   ...:     print("####################################################")
   ...:
   ...:     m = Munkres()
   ...:     indices = m.compute(costs)
   ...:     best = tuple([v[1] for v in indices])
   ...:     print("Munkres Solution:")
   ...:     print(best)
   ...:     print("Munkres Cost:")
   ...:     print(cost(best, costs))
   ...:     print()
   ...:
   ...:     print("#####################################")
   ...:     print("Local Search / Optimization Techniques")
   ...:     print("#####################################")
   ...:
   ...:     initial = random_assignment(n)
   ...:     problem = LocalAssignmentProblem(initial, initial_cost=cost(initial,␣
↪costs),
   ...:                                      extra=(costs,))
   ...:     print("Initial Assignment (randomly generated):")
   ...:     print(initial)
   ...:     print("Initial Assignment Cost:")
   ...:     print(problem.initial.cost())
   ...:     print()
   ...:
   ...:     def local_beam_width2(problem):
   ...:         return local_beam_search(problem, beam_width=2)
   ...:
   ...:     def greedy_annealing(problem):
   ...:         num_neighbors = (n * (n-1)) // 2
   ...:         return simulated_annealing(problem, initial_temp=0,
   ...:                                    temp_length=num_neighbors)
   ...:
   ...:     def annealing(problem):
   ...:         num_neighbors = (n * (n-1)) // 2
   ...:         return simulated_annealing(problem, initial_temp=1.5,
   ...:                                    temp_length=num_neighbors)
   ...:     compare_searches(problems=[problem],
   ...:                      searches=[hill_climbing, local_beam_width2,
   ...:                                annealing, greedy_annealing])
   ...:
   ...:     print()
   ...:     print("############################")
   ...:     print("Informed Search Techniques")
   ...:     print("############################")
   ...:
```

```
   ...:     # TREE SEARCH APPROACH
   ...:     empty = tuple([None for i in range(len(costs))])
   ...:     unassigned = [i for i in range(len(costs))]
   ...:
   ...:     new_costs = [[c - min(row) for c in row] for row in costs]
   ...:     min_c = [min([row[c] for row in costs]) for c,v in enumerate(costs[0])]
   ...:     new_costs = [[v - min_c[c] for c, v in enumerate(row)] for row in costs]
   ...:
   ...:     tree_problem = AssignmentProblem(empty, extra=(costs, unassigned))
   ...:
   ...:     def beam_width2(problem):
   ...:         return beam_search(problem, beam_width=2)
   ...:
   ...:     print()
   ...:     compare_searches(problems=[tree_problem],
   ...:                      searches=[beam_width2,
   ...:                                best_first_search])
   ...:
##################################################
Optimial solution using Munkres/Hungarian Algorithm
##################################################
Munkres Solution:
(2, 1, 4, 0, 7, 5, 3, 6)
Munkres Cost:
-8.58705096362
()
#####################################
Local Search / Optimization Techniques
#####################################
Initial Assignment (randomly generated):
(4, 5, 6, 1, 3, 2, 7, 0)
Initial Assignment Cost:
2.42410175219
()
Problem                 Search Alg           Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated    Solution Cost    Runtime
----------------------  -----------------  ------------  ----------------  -----------
↪------  ---------------  ---------
LocalAssignmentProblem  hill_climbing                15               196            ␣
↪     0           -7.72      0.0021
LocalAssignmentProblem  local_beam_width2            14               392            ␣
↪     0           -8.587     0.0039
LocalAssignmentProblem  annealing                   257              1092            ␣
↪     0           -8.587     0.0143
LocalAssignmentProblem  greedy_annealing              9               224            ␣
↪     0           -7.72      0.0023
()
############################
Informed Search Techniques
############################
()
Problem            Search Alg           Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated    Solution Cost    Runtime
-----------------  -----------------  ------------  ----------------  -----------------
↪-  ---------------  ---------
AssignmentProblem  beam_width2                  16               344                ␣
↪341           -8.121     0.0096
AssignmentProblem  best_first_search          1481             31949                ␣
↪21841           -8.587     1.1232
```

Graph Partition Optimization Example

```
In [1]: from py_search.problems.graph_partition import generate_graph
   ...: from py_search.problems.graph_partition import random_partition
   ...: from py_search.problems.graph_partition import LocalGraphPartitionProblem
   ...: from py_search.problems.graph_partition import cutsize
   ...: from py_search.optimization import simulated_annealing
   ...: from py_search.optimization import hill_climbing
   ...: from py_search.utils import compare_searches
   ...:
   ...: n = 10
   ...: p = 5 / (n-1)
   ...: print(n, p)
   ...: V, E = generate_graph(n, p)
   ...: initial = random_partition(V)
   ...: cost = cutsize(E, initial)
   ...:
   ...: print("########################################")
   ...: print("Local Search / Optimization Techniques")
   ...: print("########################################")
   ...:
   ...: problem = LocalGraphPartitionProblem(initial, initial_cost=cost,
   ...:                                       extra=(V,E))
   ...: print("Initial Partition Cost:")
   ...: print(cost)
   ...: print()
   ...:
   ...: def annealing(problem):
   ...:     size = (n * (n//2)) // 2
   ...:     return simulated_annealing(problem, initial_temp=5.5,
   ...:                                temp_length=size)
   ...:
   ...: def greedy_annealing(problem):
   ...:     size = (n * (n//2)) // 2
   ...:     return simulated_annealing(problem, initial_temp=0,
   ...:                                 temp_length=size)
   ...:
   ...: compare_searches(problems=[problem],
   ...:                  searches=[hill_climbing, annealing,
   ...:                            greedy_annealing])
   ...:
(10, 0)
########################################
Local Search / Optimization Techniques
########################################
Initial Partition Cost:
0
()
Problem                     Search Alg          Goal Tests    Nodes Expanded    Nodes␣
↪Evaluated    Solution Cost    Runtime
--------------------------  ----------------  ------------  ----------------  --------
↪---------  ---------------  ---------
LocalGraphPartitionProblem  hill_climbing              26                25         ␣
↪         0                0     0.0002
LocalGraphPartitionProblem  annealing                 126               125         ␣
↪         0                0     0.0016
```

```
LocalGraphPartitionProblem  greedy_annealing          126              125
↪        0               0     0.0011
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

py_search.base, 5
py_search.informed, 10
py_search.optimization, 11
py_search.problems.assignment_problem, 15
py_search.problems.eight_puzzle, 13
py_search.problems.graph_partition, 16
py_search.problems.nqueens, 14
py_search.uninformed, 8

Index

A

AnnotatedProblem (class in py_search.base), 5
AssignmentProblem (class in py_search.problems.assignment_problem), 15

B

beam_search() (in module py_search.informed), 10
best_first_search() (in module py_search.informed), 10
branch_and_bound() (in module py_search.optimization), 11
breadth_first_search() (in module py_search.uninformed), 8

C

clear() (py_search.base.PriorityQueue method), 7
copy() (py_search.problems.eight_puzzle.EightPuzzle method), 13
copy() (py_search.problems.nqueens.nQueens method), 14
cost() (in module py_search.problems.assignment_problem) 15
cost() (py_search.base.Node method), 7
cutsize() (in module py_search.problems.graph_partition), 16

D

depth() (py_search.base.Node method), 7
depth_first_search() (in module py_search.uninformed), 9

E

EightPuzzle (class in py_search.problems.eight_puzzle), 13
EightPuzzleProblem (class in py_search.problems.eight_puzzle), 13
executeAction() (py_search.problems.eight_puzzle.EightPuzzle method), 13
extend() (py_search.base.Fringe method), 6

F

FIFOQueue (class in py_search.base), 6
Fringe (class in py_search.base), 6

G

generate_graph() (in module py_search.problems.graph_partition), 16
goal_test() (py_search.base.AnnotatedProblem method), 5
goal_test() (py_search.base.Problem method), 8
goal_test() (py_search.problems.assignment_problem.AssignmentProblem method), 15
goal_test() (py_search.problems.assignment_problem.LocalAssignmentProl method), 15
goal_test() (py_search.problems.eight_puzzle.EightPuzzleProblem method), 13
goal_test() (py_search.problems.graph_partition.LocalGraphPartitionProble method), 16
goal_test() (py_search.problems.nqueens.LocalnQueensProblem method), 14
goal_test() (py_search.problems.nqueens.nQueensProblem method), 14
graph_search() (in module py_search.uninformed), 9

H

hill_climbing() (in module py_search.optimization), 11

I

iterative_deepening_best_first_search() (in module py_search.informed), 10
iterative_deepening_search() (in module py_search.uninformed), 9

L

legalActions() (py_search.problems.eight_puzzle.EightPuzzle method), 13
LIFOQueue (class in py_search.base), 6
local_beam_search() (in module py_search.optimization), 12

LocalAssignmentProblem (class in py_search.problems.assignment_problem), 15
LocalGraphPartitionProblem (class in py_search.problems.graph_partition), 16
LocalnQueensProblem (class in py_search.problems.nqueens), 14

M

min_cost_heuristic() (py_search.problems.assignment_problem.AssignmentProblem method), 15
misplaced_tile_heuristic() (py_search.problems.eight_puzzle.EightPuzzleProblem method), 13

N

Node (class in py_search.base), 6
node_value() (py_search.base.AnnotatedProblem method), 5
node_value() (py_search.base.Problem method), 8
node_value() (py_search.problems.assignment_problem.AssignmentProblem method), 15
node_value() (py_search.problems.assignment_problem.LocalAssignmentProblem method), 15
node_value() (py_search.problems.eight_puzzle.EightPuzzleProblem method), 14
node_value() (py_search.problems.eight_puzzle.NoHeuristic method), 14
node_value() (py_search.problems.graph_partition.LocalGraphPartitionProblem method), 16
node_value() (py_search.problems.nqueens.nQueensProblem method), 14
NoHeuristic (class in py_search.problems.eight_puzzle), 14
nQueens (class in py_search.problems.nqueens), 14
nQueensProblem (class in py_search.problems.nqueens), 14
num_conflicts() (py_search.problems.nqueens.nQueens method), 14

P

path() (py_search.base.Node method), 7
peek() (py_search.base.PriorityQueue method), 8
peek_value() (py_search.base.PriorityQueue method), 8
pop() (py_search.base.FIFOQueue method), 6
pop() (py_search.base.Fringe method), 6
pop() (py_search.base.LIFOQueue method), 6
pop() (py_search.base.PriorityQueue method), 8
print_matrix() (in module py_search.problems.assignment_problem), 15
PriorityQueue (class in py_search.base), 7
Problem (class in py_search.base), 8
push() (py_search.base.FIFOQueue method), 6
push() (py_search.base.Fringe method), 6
push() (py_search.base.PriorityQueue method), 8
py_search.base (module), 5
py_search.informed (module), 10
py_search.optimization (module), 11
py_search.problems.assignment_problem (module), 15
py_search.problems.eight_puzzle (module), 13
py_search.problems.graph_partition (module), 16
py_search.problems.nqueens (module), 14
py_search.uninformed (module), 8

R

random() (in module py_search.optimization), 12
random() (in module py_search.problems.graph_partition), 16
random_assignment() (in module py_search.problems.assignment_problem), 15
random_matrix() (in module py_search.problems.assignment_problem), 16
random_node() (py_search.base.AnnotatedProblem method), 5
random_node() (py_search.base.Problem method), 8
random_node() (py_search.problems.assignment_problem.LocalAssignmen method), 15
random_node() (py_search.problems.graph_partition.LocalGraphPartitionP method), 16
random_node() (py_search.problems.nqueens.LocalnQueensProblem method), 14
random_partition() (in module py_search.problems.graph_partition), 16
random_successor() (py_search.base.AnnotatedProblem method), 5
random_successor() (py_search.base.Problem method), 8
random_successor() (py_search.problems.assignment_problem.LocalAssign method), 15
random_successor() (py_search.problems.graph_partition.LocalGraphPartit method), 16
random_successor() (py_search.problems.nqueens.LocalnQueensProblem method), 14
randomize() (py_search.problems.eight_puzzle.EightPuzzle method), 13
randomize() (py_search.problems.nqueens.nQueens method), 14
remove() (py_search.base.FIFOQueue method), 6

S

simulated_annealing() (in module py_search.optimization), 12
successors() (py_search.base.AnnotatedProblem method), 6
successors() (py_search.base.Problem method), 8

successors() (py_search.problems.assignment_problem.AssignmentProblem method), 15
successors() (py_search.problems.assignment_problem.LocalAssignmentProblem method), 15
successors() (py_search.problems.eight_puzzle.EightPuzzleProblem method), 14
successors() (py_search.problems.graph_partition.LocalGraphPartitionProblem method), 16
successors() (py_search.problems.nqueens.LocalnQueensProblem method), 14
successors() (py_search.problems.nqueens.nQueensProblem method), 15

T

tree_search() (in module py_search.uninformed), 9

U

update_cost_limit() (py_search.base.PriorityQueue method), 8

W

widening_beam_search() (in module py_search.informed), 11