
Decision Theoretic Troubleshooting

Ariana Carnielli, Ivan Kachaikin

juin 05, 2020

Contents:

Index des modules Python

19


```
class DecisionTheoreticTroubleshooting.TroubleShootingProblem (bayesian_network,  
                                                             costs,  
                                                             nodes_types)
```

Classe créée pour représenter un problème de Troubleshooting. Contient des méthodes divers pour résoudre le problème. Utilise le module pyAgrum pour manipuler le réseau bayésien utilisé pour représenter le problème. Les noeuds du réseau bayésien sont référencés par des strings.

Paramètres

- **bayesian_network** (*pyAgrum.BayesNet*) – Représente le réseau bayésien (BN) modélisant un problème donné.
- **costs** (*list(dict)*) – Liste avec deux dictionnaires, le premier avec les coûts de réparation (exactes ou avec des minimum/maximun) et le deuxième avec les coûts d’observation des noeuds.
- **nodes_types** (*dict*) – Dictionnaire où les clés représentent les noeuds du BN et les valeurs leurs types associés (set de string).

bayesian_network

Représente le réseau bayésien (BN) qui modélise un problème donné.

Type *pyAgrum.BayesNet*

bay_lp

Fait l’inference exacte pour le BN passé en argument.

Type *pyAgrum.LazyPropagation*

costs_rep

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs leurs coûts de reparation (float).

Type *dict*

costs_rep_interval

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs des listes avec les coûts minimum et maximum de reparation (floats).

Type *dict*

costs_obs

Dictionnaire de coûts où les clés représentent les noeuds du BN et les valeurs leurs coûts d’observation (float).

Type *dict*

repairable_nodes

Ensemble de noeuds qui correspondent aux éléments du système concerné qui peuvent être réparés.

Type *set*

unrepairable_nodes

Ensemble de noeuds qui correspondent aux éléments d’un système qui ne peuvent pas être réparés.

Type *set*

problem_defining_node

Noeud qui représente le problème à être réglé (système fonctionnel ou pas).

Type *string*

observation_nodes

Ensemble de noeuds qui correspondent aux éléments du système qui peuvent être observés.

Type *set*

service_node

Noeud qui représente l’appel au service (appel à la réparation sûre du système).

Type *string*

evidences

Dictionnaire où les clés représentent les éléments du système qui ont des evidences modifiés dans `bay_lp` (donc qui ont été réparés/observés) et les valeurs sont les inferences faites.

Type `dict`

ECR_ECO_wrapper (*debug=False*)

Calcule l'ECR myope pour chaque prochaine « observation-réparation » possible et l'ECO pour chaque prochaine observation globale possible.

Paramètres `debug` (*bool, facultatif*) – Si `True`, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **chosen_node** (*string*) – Noeud choisi.
- **type_node** (*string*) – Type du noeud choisi (« repair » ou « obs »).
- **list_ecr** (*list(tuple)*) – ECRs des noeuds d'« observation-réparation ».
- **list_eco** (*list(tuple)*) – ECOs des noeuds d'observation globale.

_compute_costs (*costs*)

Prend en argument un dictionnaire de couts qui peut avoir des valeurs exactes ou des intervalles de valeurs (de la forme [minimum, maximum]) et le transforme en 2 dictionnaires, un avec les esperances de cout pour chaque clé et l'autre avec des intervalles de valeurs pour chaque clé.

Paramètres `costs` (*dict*) – Dictionnaire de couts où les clés représentent les noeuds du BN et les valeurs sont de nombres ou de listes de deux nombres.

Renvoie

- **expected_cost** (*dict*) – Dictionnaire où les clés représentent les noeuds du BN et les valeurs l'esperance de cout de ce noeud. Si la valeur initiale était déjà un nombre, ce nombre est seulement copié, sinon on considère que la valeur est une variable aléatoire avec une distribution uniforme dans l'intervalle et donc l'esperance est la moyenne des extremités de l'intervalle.
- **interval_cost** (*dict*) – Dictionnaire où les clés représentent les noeuds du BN et les valeurs sont des listes contenant les deux extremités des intervalles dans lesquels les couts se trouvent. Si la valeur initiale était déjà un nombre, ce nombre est copié comme les deux extremités. Si la valeur initiale était un iterable, on le transforme en liste.

_create_nodes (*names, rep_string='_repair', obs_string='_observation', obs_rep_couples=False*)

Crée des noeuds de StrategyTree à partir de leurs noms dans le réseau Bayésien.

Paramètres

- **names** (*list(str)*) – Noms des noeuds de réparations/observations/ observations-réparations dans le réseau Bayésien à partir desquels on crée les noeuds.
- **rep_string** (*string, facultatif*) – Dans le cas où on ne considère pas des couples, on utilise ce paramètre comme un suffixe pour les noeuds de réparation pour les séparer de ceux d'observation.
- **obs_string** (*string, facultatif*) – Suffixe pour les noeuds d'observation.
- **obs_rep_couples** (*bool, facultatif*) – Variable booléenne qui indique si on suppose l'existence de couples « observation-réparation » dans l'arbre de stratégie.

Renvoie `nodes` – Liste de noeuds créés.

Type renvoyé `list(StrategyTree.NodeST)`

_evaluate_all_st (*feasible_nodes, obs_next_nodes=None, parent=None, fn_immutable=None, debug_nb_call=0, debug_iter=False, debug_st=False, obs_rep_couples=False, obs_obsolete=False, sock=None*)

Méthode récursive qui trouve le meilleur arbre de stratégie étant donné une configuration du problème en dénombrant tous les arbres admissibles.

Paramètres

- **feasible_nodes** (*list(StrategyTree.NodeST)*) – Liste des noeuds admissibles qu'on a le droit d'utiliser pour construire l'arbre.

- **obs_next_nodes** (*list(list(str)), facultatif*) – Pile des attributs des arcs qui partent des noeuds déjà utilisés.
- **parent** (*list(tuple(StrategyTree.NodeST, StrategyTree.NodeST, StrategyTree.StrategyTree))*, facultatif) – Pile des *parents* vers lesquels il faudra se retourner quand on remplit entièrement la branche courante.
- **fn_immutable** (*list(list(StrategyTree.NodeST))*, facultatif) – Pile des noeuds admissibles qu'on peut utiliser pour les branches différentes qui suivent un noeud d'observation.
- **debug_nb_call** (*int, facultatif*) – Profondeur de la récursivité.
- **debug_iter** (*bool, facultatif*) – Indique s'il faut afficher l'index de l'itération de l'appel.
- **debug_st** (*bool, facultatif*) – Précise s'il faut afficher tous les arbres intermédiaires.
- **obs_rep_couples** (*bool, facultatif*) – Indique si on suppose l'existence de couples « observation-réparation » dans l'arbre de stratégie soumis.
- **obs_obsolete** (*bool, facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. qu'une observation devient obsolète en réparant une composante.
- **sock** (*socket.socket, facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

_expected_cost_of_repair_internal (*strategy_tree, evid_init=None, prob=1.0, obs_obsolete=False*)

Partie récursive de la fonction `expected_cost_of_repair`.

Paramètres

- **strategy_tree** (*StrategyTree.StrategyTree*) – Arbre de stratégie dont le coût il faut calculer.
- **evid_init** (*dict(str: str), facultatif*) – Dictionnaire d'évidences utilisé dans les appels récursifs.
- **prob** (*float, facultatif*) – Probabilité initiale.
- **obs_obsolete** (*bool, facultatif*) – Si True, on remet en cause les noeuds d'observation globale après une réparation.

Renvoie `ecr` – Coût espéré de réparation d'un arbre de stratégie fourni.

Type renvoyé `float`

_next_node_id ()

Permet d'obtenir la prochaine valeur de *id* pour un noeud de `StrategyTree` à partir des ids qu'on utilise déjà stockés dans `self._nodes_ids_db_brute_force`.

Renvoie `next_id` – Prochain id qu'on peut utiliser avec garantie qu'il n'existe pas de noeuds déjà avec le même id dans l'arbre de stratégie courant.

Type renvoyé `str`

_start_bay_lp ()

Ajoute des inférences vides aux noeuds du BN qui peuvent être modifiés (réparés/observés/appelés). Ces évidences ne changent pas les probabilités, elles servent pour qu'on puisse utiliser la méthode `chgEvidence` de `pyAgrum` à la suite.

add_evidence (*node, evidence*)

Fonction wrapper pour la fonction `chgEvidence` de l'objet `bay_lp` du type `pyAgrum.LazyPropagation` qui additionne une inference et maintient le dictionnaire `evidences` actualisé. L'evidence passé en argument ne doit pas être une evidence « vide » (des 1, utilisé plutôt la fonction `remove_evidence`).

Paramètres

- **node** (*string*) – Nom du noeud de `bay_lp` qui va être modifié.
- **evidence** – Nouvelle inference pour le noeud traité (généralement une string ici, cf. les types acceptés par `chgEvidence`)

best_EVOI ()

Détermine la composante qui a la plus grande valeur espérée d'information (EVOI) correspondant à avoir plus d'information sur l'intervalle de valeur de son cout.

Renvoie Le nom du noeud de réparation avec la plus grande EVOI et la valeur d'EVOI correspondante.

Type renvoyé tuple(string, float)

brute_force_solver (*debug=False, mode='all', obs_rep_couples=False, obs_obsolete=False, sock=None*)

Wrapper des différents algorithmes de recherche exhaustive qui calcule la solution exacte optimale étant donné un problème de Troubleshooting.

Paramètres

- **debug** (*bool / tuple, facultatif*) – Indique s'il faut afficher les résultats intermédiaires du calcul. Un tuple avec `len(tuple) == 2` ou un booléen (équivalent au cas où on passe un tuple avec deux valeurs identiques). Le premier composant indique s'il faut afficher l'index de l'itération tandis que le deuxième précise s'il faut afficher tous les arbres intermédiaires.
- **mode** (*str, facultatif*) – Mode de calcul : soit "dp" pour la programmation dynamique, soit "all" pour le dénombrement complet.
- **obs_rep_couples** (*bool, facultatif*) – Indique si on suppose des couples « observation-réparation » dans l'arbre de stratégie.
- **obs_obsolete** (*bool, facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. si c'est possible qu'une observation devient obsolète en réparant une composante.
- **sock** (*socket.socket, facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

Renvoie

- **best_st** (*StrategyTree.StrategyTree*) – Le meilleur arbre de stratégie trouvé.
- **best_ecr** (*float*) – Le coût espéré de réparation du meilleur arbre trouvé i.e. $ECR(best_st)$.

brute_force_solver_actions_only (*debug=False*)

Cherche une séquence optimale de réparation par une recherche exhaustive en choisissant la séquence de meilleur ECR. Pour le cas où on ne considère que les actions de réparation il suffit de dénombrer toutes les permutations possibles d'un ensemble des actions admissibles.

Paramètres **debug** (*bool, facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **min_seq** (*list(str)*) – Séquence optimale trouvée dont le coût est le plus petit possible.
- **min_ecr** (*float*) – Coût espéré de réparation correspondant à min_seq.

brute_force_solver_tester (*true_prices, epsilon, nb_min=100, nb_max=200, strategy_tree=None, mode='dp', obs_rep_couples=False, true_prices_obs=None*)

Test empirique de la méthode `brute_force_solver`, ou bien de l'arbre de stratégie obtenu par l'algorithme. Le mécanisme selon lequel on teste la stratégie est exactement le même que celui utilisé au-dessus dans les autres testers.

Paramètres

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être réalisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être réalisées.
- **strategy_tree** (*StrategyTree.StrategyTree, facultatif*) – Arbre de stratégie qu'il faut tester; si rien passé, on calcule l'arbre avec la méthode `brute_force_solver`.

- **mode** (*string*, *facultatif*) – Paramètre à passer à la méthode `brute_force_solver` si on doit l'exécuter. Peut être égal à "all" pour le dénombrement complet ou "dp" pour la programmation dynamique.
- **obs_rep_couples** (*bool*, *facultatif*) – Paramètre à passer à la méthode `brute_force_solver` si on doit l'exécuter. Indique si on suppose l'existence de couples « observation-réparation » dans l'arbre de stratégie.
- **true_prices_obs** (*dict*, *facultatif*) – Dictionnaire de prix d'observations des composantes observables.

Renvoie

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
- **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.

compute_EVOIs ()

Calcule les valeurs espérées d'information (EVOIs) correspondant à avoir plus d'information sur l'intervalle de valeur des couts de réparation pour chaque composante réparable.

Renvoie **evoi** – Dictionnaire indexé par les noeuds réparables contenant la valeur d'une information plus précise du cout de réparation de ces noeuds.

Type renvoyé dict

draw_true_prices ()

Tire au hasard des prix de réparation selon des lois uniformes sur les intervalles stockés dans `self.costs_rep_interval`.

Renvoie Dictionnaire avec prix de réparation.

Type renvoyé dict

dynamic_programming_solver (*feasible_nodes=None*, *evidence=None*, *debug_iter=False*, *debug_st=False*, *obs_rep_couples=False*, *prob=1.0*, *obs_obsolete=False*, *sock=None*, *debug_nb_call=0*)

Méthode récursive qui trouve le meilleur arbre de stratégie étant donné une configuration du problème en utilisant comme approche la programmation dynamique, supposant qu'un sous-arbre de l'arbre optimal est lui-même également optimal.

Paramètres

- **feasible_nodes** (*list(StrategyTree.NodeST)*, *facultatif*) – Liste des noeuds admissibles qu'on a le droit d'utiliser pour construire l'arbre.
- **evidence** (*dict*, *facultatif*) – Dictionnaire des évidences initiales pour un appel de cette fonction.
- **debug_iter** (*bool*, *facultatif*) – Indique s'il faut afficher l'index de l'itération dans l'appel.
- **debug_st** (*bool*, *facultatif*) – Précise s'il faut afficher tous les arbres intermédiaires.
- **obs_rep_couples** (*bool*, *facultatif*) – Indique si on suppose l'existence de couples « observation-réparation » dans l'arbre de stratégie soumis.
- **prob** (*float*, *facultatif*) – Probabilité que le système ne marche toujours pas.
- **obs_obsolete** (*bool*, *facultatif*) – Indique si on suppose la possibilité des « observations obsolètes », i.e. qu'une observation devient obsolète en réparant une composante.
- **sock** (*socket.socket*, *facultatif*) – Socket de communication entre le processus qui effectue le calcul et celui qui met à jour l'interface. Paramètre nécessaire pour que le ProgressBar de l'interface marche proprement.

— **debug_nb_call** (*int*, *facultatif*) – Profondeur de la récursivité.

elicitation (*noeud*, *islower*)

Met à jour l'intervalle de valeurs de cout pour le noeud et son espérance en fonction de la réponse de l'utilisateur.

Paramètres

- **noeud** (*string*) – Nom du noeud à mettre à jour.
- **islower** (*bool*) – Représente la réponse à la question : Est-ce que le cout est plus petit que l'espérance courante ?

elicitation_solver_tester (*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*, *debug=False*)

Test empirique de la résolution avec élicitation. À chaque fois qu'on doit prendre une action, on vérifie d'abord s'il y a des questions à répondre et, si oui, on les répond toutes correctement selon *true_prices*. Ensuite, la méthode calcule la séquence d'actions itérativement à l'aide de *myopic_solver* et réalise au plus *nb_max* répétitions d'un système tiré au hasard, le tirage au hasard étant identique à celui de *myopic_solver_tester*. Si après *nb_min* répétitions l'erreur estimée est plus petite que *epsilon*, on fait une sortie anticipée. La fonction calcule les couts empiriques de réparation, en utilisant pour cela *true_prices*.

Paramètres

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être réalisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être réalisées.
- **debug** (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
- **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.
- **cpt_questions** (*numpy.ndarray*) – Tableau avec le nombre de questions répondues à chaque répétition.

expected_cost_of_repair (*strategy_tree*, *obs_obsolete=False*)

Calcule le coût espéré de réparation étant donné un arbre de décision.

Paramètres

- **strategy_tree** (*StrategyTree.StrategyTree*) – Arbre de stratégie dont le coût il faut calculer.
- **obs_obsolete** (*bool*, *facultatif*) – Si True, on remet en cause les noeuds d'observation globale après une réparation.

Renvoie *ecr* – Coût espéré de réparation d'un arbre de stratégie fourni.

Type renvoyé *float*

expected_cost_of_repair_seq_of_actions (*seq*)

Calcule le coût espéré de réparation à partir d'une séquence d'actions donnée. On utilise la formule ECR = $\text{coût}(C1 \mid E0) + P(C1 = \text{Normal} \mid E0) * \text{coût}(C2 \mid E1) + P(C1 = \text{Normal} \mid E0) * P(C2 = \text{Normal} \mid E1) * \text{coût}(C2 \mid E2) + \dots$

Paramètres *seq* (*list(str)*) – Séquence d'actions de réparations dont le coût espéré est à calculer.

Renvoie *ecr* – Coût espéré de réparation de la séquence donnée.

Type renvoyé *float*

get_proba (*node*, *value*)

Récupère à partir du réseau bayésien la probabilité que le noeud *node* ait la valeur *value*.

Paramètres

- **node** (*string*) – Nom du noeud de bay_lp dont on veut calculer la probabilité.
- **value** (*string*) – Valeur du noeud dont on veut calculer la probabilité.

Renvoie La probabilité $P(\text{node} = \text{value})$

Type renvoyé float

myopic_solver (*debug=False*, *esp_obs=False*)

Implémente une étape du solveur myope. Étant donné l'état actuel du réseau, ce solveur utilise dans un premier temps le `simple_solver_obs` pour déterminer quelle action du type « observation-réparation » serait la meilleure. Ensuite, il calcule les coûts myopes espérés avec chaque observation possible et choisit à la fin la meilleure action à être prise.

Cette fonction est itérative et ne fait qu'un seul tour de l'algorithme myope car elle attend des nouvelles informations venues de l'utilisateur (résultat de l'observation si c'est le cas).

Paramètres

- **debug** (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.
- **esp_obs** (*bool*, *facultatif*) – Si True, retourne en plus un dictionnaire indexé par les observations possibles et contenant leurs coûts myopes espérés respectifs.

Renvoie

- **chosen_node** (*string*) – Le meilleur noeud de ce tour
- **type_node** (*string*) – Type du meilleur noeud (« repair » ou « obs »)
- **eco** (*dict*) – Retourné uniquement lorsque `esp_obs` vaut True. Dictionnaire des coûts espérés des observations.

myopic_solver_st (*evid_init=None*)

Une méthode qui récupère un arbre de stratégie qu'on peut construire à partir de `myopic_solver`.

Paramètres **evid_init** (*dict*, *facultatif*) – Un dictionnaire des évidences par défaut.

Renvoie **strat_tree** – Un arbre de stratégie qu'on construit utilisant pas-à-pas une méthode `myopic_solver` en remplissant cet arbre en largeur.

Type renvoyé *StrategyTree.StrategyTree*

myopic_solver_tester (*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*, *debug=False*)

Test empirique de la méthode `myopic_solver`. Cette méthode calcule la séquence d'actions itérativement à l'aide de `myopic_solver` et réalise au plus `nb_max` répétitions d'un système tiré au hasard. À chaque observation globale, son résultat est tiré au hasard. Pour les paires « observation-réparation », on tire au hasard si la composante correspondante marche ou pas. Si oui, on ajoute juste le coût de l'observation et on continue, si non, on ajoute les coûts d'observation et de réparation et on s'arrête (single fault assumption). Si on a une réparation simple sans observation associée, on ajoute directement le coût de réparation de la composante. Si après `nb_min` répétitions l'erreur estimée est plus petite que `epsilon`, on fait une sortie anticipée. La fonction calcule les coûts empiriques de réparation, en utilisant pour cela `true_prices`.

Paramètres

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être réalisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être réalisées.
- **debug** (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les coûts associés.
- **mean** (*float*) – Moyenne des coûts.

- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.
- **cpt_obs** (*numpy.ndarray*) – Tableau avec le nombre d'observations globales faites à chaque répétition.

myopic_wrapper (*debug=False*)

Interface textuelle pour le solveur myope. Utilise `myopic_solver` à chaque tour de boucle pour déterminer la meilleure action à prendre. Si c'est une observation, le résultat de l'observation est demandé, sinon on demande juste si l'action a résolu le problème. Les élicitations de couts ne sont pas implémentées. Les entrées de l'utilisateur ne sont pas sécurisées.

Paramètres **debug** (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

noeud_ant (*node*, *visites*)

Détermine tous les noeuds d'observation impactés par un changement du noeud `node` et qui sont antécédents de `node`, sans visiter les noeuds déjà dans l'ensemble des visites. Cette fonction est auxiliaire et n'a pas vocation à être appelée en dehors de la fonction principale `observation_obsolete`.

Paramètres

- **node** (*string*) – Nom du noeud dont l'information a changé.
- **visites** (*set*) – Contient les noeuds déjà visités.

Renvoie **ant_obs** – Ensemble des noeuds d'observation affectés par `node` et qui sont antécédents de `node` sans être dans `visites`.

Type renvoyé *set*

observation_obsolete (*node*)

Étant donné un noeud dont l'information a changé, on détermine, à partir du réseau bayésien, tous les noeuds d'observation impactés par ce changement.

Paramètres **node** (*string*) – Nom du noeud dont l'information a changé.

Renvoie **obs** – Ensemble contenant les noeuds d'observation impactés.

Type renvoyé *set*

remove_evidence (*node*)

Fonction wrapper pour la fonction `chgEvidence` de l'objet `bay_lp` du type `pyAgrum.LazyPropagation` qui retire une inférence et maintient le dictionnaire `evidences` actualisé.

Paramètres **node** (*string*) – Nom du noeud de `bay_lp` qui va être modifié.

reset_bay_lp (*dict_inf={}*)

Reinitialise les inférences des noeuds du BN qui peuvent être modifiés (réparés/observés/appelés). Pour les noeuds dans `dict_inf`, l'inférence est mise à la valeur associée au noeud dans `dict_inf`, pour les autres l'inférence est mise à 1.

Paramètres **dict_inf** (*dict*, *facultatif*) – Dictionnaire où les clés sont des noeuds et les valeurs sont des inférences.

simple_solver (*debug=False*)

Solveur simple pour le problème du TroubleShooting. On ne prend pas en considération des observations et on ne révisé pas les probabilités, c'est-à-dire on ne met pas à jour les probabilités si on répare une composante. À cause de cela, ce solveur n'est pas interactif et renvoie l'ordre de réparation entière (jusqu'au appel au service).

Paramètres **debug** (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **rep_seq** (*list*) – Séquence des noeuds à être réparés dans l'ordre.
- **exp_cost** (*float*) – Espérance du coût de réparation de cette séquence.

simple_solver_obs (*debug=False*)

Solveur simple pour le problème du Troubleshooting. On prend en considération des paires « observation-réparation » (cf. définition dans l'état de l'art) mais pas les observations globales et on révisé les probabilités, c'est-à-dire on met à jour les probabilités quand on « répare » une composante avant de calculer la prochaine composante de la séquence.

Le solveur n'est pas encore interactif et renvoie l'ordre de réparation entière (jusqu'au appel au service).

Cette choix à été fait car on utilise cet algorithme comme part de l'agorithme plus complexe et iteratif.

Paramètres *debug* (*bool*, *facultatif*) – Si True, affiche des messages montrant le déroulement de l'algorithme.

Renvoie

- **rep_seq** (*list*) – Séquence des noeuds à être réparés dans l'ordre.
- **exp_cost** (*float*) – Espérance du coût de réparation de cette séquence.

simple_solver_obs_tester (*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*)

Test empirique de la méthode simple_solver_obs. Cette méthode calcule la séquence d'actions à l'aide de simple_solver_obs et réalise au plus nb_max répétitions d'un système tiré au hasard : si on a une paire « observation-réparation », on tire au hasard si la composante correspondante marche ou pas. Si oui, on ajoute juste le cout de l'observation et on continue, si non, on ajoute les couts d'observation et de réparation et on s'arrête (single fault assumption). Si on a une réparation simple sans observation associée, on ajoute directement le cout de réparation de la composante. Si après nb_min répétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule les couts empiriques de réparation, en utilisant pour cela true_prices.

Paramètres

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être réalisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être réalisées.

Renvoie

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.

simple_solver_tester (*true_prices*, *epsilon*, *nb_min=100*, *nb_max=200*)

Test empirique de la méthode simple_solver. Cette méthode calcule la séquence d'actions à l'aide de simple_solver et réalise au plus nb_max répétitions d'un système tiré au hasard : à chaque fois qu'on a une probabilité qu'une action résoud le problème, on tire au hasard pour déterminer si le problème a effectivement été résolu ou pas suite à cette action. Si après nb_min répétitions l'erreur estimée est plus petite que epsilon, on fait une sortie anticipée. La fonction calcule aussi les couts empiriques de réparation, en utilisant pour cela true_prices. Cette méthode utilise la single fault assumption.

Paramètres

- **true_prices** (*dict*) – Dictionnaire de prix de réparation des composantes réparables.
- **epsilon** (*float*) – Tolerance relative de la moyenne.
- **nb_min** (*int*) – Nombre minimum de répétitions à être réalisées.
- **nb_max** (*int*) – Nombre maximum de répétitions à être réalisées.

Renvoie

- **sortie_anti** (*bool*) – True en cas de sortie anticipée, False sinon.
- **costs** (*numpy.ndarray*) – Tableau avec les cout associés.
- **mean** (*float*) – Moyenne des couts.
- **std** (*float*) – Variance des couts.
- **cpt_repair** (*numpy.ndarray*) – Tableau avec le nombre de composantes réparées à chaque répétition.

class DecisionTheoreticTroubleshooting.**bcolors**

Stockage de couleurs.

DecisionTheoreticTroubleshooting.**diff_dicts** (*left*, *right*)

Calcule la différence des dictionnaires *left* et *right* : les entrées de *left* dont la clé est aussi présente dans *right* sont supprimées, les autres sont gardées.

Paramètres

- **left** (*dict*) – Premier dictionnaire, duquel on supprime les clés apparaissant dans *right*.
- **right** (*dict*) – Deuxième dictionnaire, celui avec les clés qui doivent être supprimées de *left*.

Renvoie **res** – Résultat de la différence entre *left* et *right*.

Type renvoyé dict

DecisionTheoreticTroubleshooting.**merge_dicts** (*left*, *right*)

Fusionne deux dictionnaire passés sans les changer. Les couples (clé, valeur) du dictionnaire *right* sont plus prioritaires que celles de *left* ; c'est-à-dire, s'il existe une valeur associée à la même clé *k* dans les deux dictionnaires, on ajoute dans le résultat seulement celle qui appartient à *right*.

Paramètres

- **left** (*dict*) – Un des dictionnaires à fusionner, celui qui est moins prioritaire.
- **right** (*dict*) – L'autre dictionnaire à fusionner, celui qui est plus prioritaire.

Renvoie **res** – Résultat de la fusion.

Type renvoyé dict

DecisionTheoreticTroubleshooting.**shallow_copy_list_of_copyable** (*l*)

Crée une copie de profondeur 1 de la liste passée en argument : la liste est recopiée et remplie avec l'appel de la méthode copy() en chaque élément de la liste donnée.

Paramètres **l** (*list<Copyable>*) – La liste qui sera copiée. Chacun de ses éléments doit implémenter la méthode copy().

Renvoie **cl** – Copie de profondeur 1 de la liste passée en argument.

Type renvoyé list<Copyable>

DecisionTheoreticTroubleshooting.**shallow_copy_parent** (*parent*)

Crée une copie superficielle de *parent* (cf la méthode TroubleShootingProblem._evaluate_all_st ci-dessous).

Paramètres **parent** (*list(tuple(StrategyTree.NodeST, StrategyTree.NodeST, StrategyTree.StrategyTree))*) – Parent dont la copie il faut créer.

Renvoie **parent_copy** – Copie superficielle du parent passé.

Type renvoyé list(tuple(StrategyTree.NodeST, StrategyTree.NodeST, StrategyTree.StrategyTree))

class StrategyTree.**NodeST** (*id*, *cost*, *name=None*)

Représente un noeud abstrait d'un arbre de stratégie ; on remarque que cette classe ne dispose pas d'attribut correspondant à des enfants (il n'y a pas un attribut qui correspond à les noeuds suivants), pourtant, on suppose que ses sous-classes en auront.

Paramètres

- **id** (*str*) – Identificateur unique d'un noeud.
- **cost** (*float*) – Correspond au « coût » du noeud.
- **name** (*str*, *facultatif*) – Nom du noeud qui peut ne pas être unique ; si rien a été soumis, on pose que *_name = _id*.

_id

Identificateur unique d'un noeud.

Type str

_cost

Correspond au « coût » du noeud.

Type float

__name

Nom du noeud qui peut ne pas être unique ; si rien a été soumis, on pose que `__name = __id`.

Type str

__eq__ (*other*)

Overloading de l'opérateur `__eq__` ; on dit que deux noeuds sont égaux ssi ils ont les mêmes ids.

Paramètres *other* (*NodeST*) – Le noeud à comparer avec le noeud concerné.

Renvoie *comp_res* – True si `self._id == other._id` ET si `self` et `other` ont le même type. False, sinon.

Type renvoyé bool

__str__ ()

Overloading de l'opérateur `__str__`.

Renvoie *corr_str* – La représentation du noeud sous la forme de str.

Type renvoyé str

add_child (*child*)

Méthode abstraite qui ajouterait un enfant au noeud.

Paramètres *child* (*NodeST*) – L'enfant à ajouter.

bn_labels_children_association ()

Méthode abstraite qui retournera un dictionnaire des associations entre les labels du réseau Bayésien et les enfants du noeud.

Renvoie *da* – Le dictionnaire des associations.

Type renvoyé dict

copy ()

Retournerait une copie superficielle du noeud.

Renvoie *copy* – Copie superficielle du noeud.

Type renvoyé *NodeST*

get_child_by_attribute (*attr*)

Méthode abstraite qui retournerait l'enfant du noeud correspondant à *attr*.

Paramètres *attr* (*str*) – L'attribut de l'enfant qu'il faut retourner.

Renvoie *child* – L'enfant du noeud concerné qui correspond à l'attribut soumis.

Type renvoyé *NodeST*

get_cost ()

Getter de l'attribut `__cost`.

Renvoie *__cost* – La valeur du coût du noeud concerné.

Type renvoyé float

get_id ()

Getter de l'attribut `__id`.

Renvoie *__id* – Identificateur unique courant du noeud concerné.

Type renvoyé str

get_list_of_children ()

Méthode abstraite qui permettrait d'obtenir la liste de tous les enfants d'un noeud.

Renvoie *list_of_children* – Liste de tous les enfants d'un noeud.

Type renvoyé list(*NodeST*)

get_name ()

Getter de l'attribut `__name`.

Renvoie `_name` – Le nom du noeud concerné.

Type renvoyé `str`

set_child_by_attribute (*attr*, *child=None*)

Méthode abstraite qui ajouterait un enfant correspondant à *attr* aux enfants du noeud concerné.

Paramètres

— **attr** (*str*) – L’attribut (un type) de l’enfant qui va être ajouté.

— **child** (*NodeST*, *facultatif*) – L’enfant qu’on veut ajouter. Il faut qu’il corresponde à l’attribut soumis.

set_cost (*cost*)

Setter de l’attribut `_cost`.

Paramètres **cost** (*float*) – Valeur du coût du noeud en question. Plus grand ou égal à zéro.

set_id (*id*)

Setter de l’attribut `_id`.

Paramètres **id** (*str*) – Nouvel identificateur du noeud en question. Doit être unique.

set_name (*name*)

Setter de l’attribut `_name`.

Paramètres **name** (*str*) – Le nouveau nom du noeud en question.

class `StrategyTree.Observation` (*id*, *cost*, *name=None*, *yes_child=None*, *no_child=None*, *obs_rep_couples=False*)

Classe pour représenter les noeuds des arbres de stratégie correspondants à des actions d’observation.

Paramètres

— **id** (*str*) – Identificateur unique du noeud.

— **cost** (*float*) – Correspond au « coût » du noeud.

— **name** (*str*, *facultatif*) – Nom du noeud qui peut ne pas être unique ; si rien a été soumis, on pose que `_name = _id`.

— **yes_child** (*NodeST*, *facultatif*) – Enfant du noeud qui correspond à la branche « yes ».

— **no_child** (*NodeST*, *facultatif*) – Enfant du noeud qui correspond à la branche « no ».

— **obs_rep_couples** (*bool*, *facultatif*) – Indique si le noeud représente un couple observation-réparation ou pas.

_id

Identificateur unique du noeud.

Type `str`

_cost

Correspond au « coût » du noeud.

Type `float`

_name

Nom du noeud qui peut ne pas être unique ; si rien a été soumis, on pose que `_name = _id`.

Type `str`, facultatif

_yes_child

Enfant du noeud qui correspond à la branche « yes ».

Type *NodeST*, facultatif

_no_child

Enfant du noeud qui correspond à la branche « no ».

Type *NodeST*, facultatif

_obs_rep_couples

Indique si le noeud représente un couple observation-réparation ou pas.

Type bool, facultatif

add_child (*child*)

Ajoute un enfant dans la liste des enfants du noeud. ATTENTION : ce méthode ne change pas un enfant qui existe déjà ; pour cela, veuillez utiliser `set_child`. Par défaut, essaie d'ajouter l'enfant au branche `_yes_child` d'abord.

Paramètres *child* (*NodeST*) – L'enfant à ajouter.

bn_labels_children_association ()

Retourne un dictionnaire des associations entre les labels d'un réseau Bayésien et les enfants du noeud.

Renvoie *da* – Le dictionnaire des associations concerné.

Type renvoyé dict

copy ()

Retourne une copie superficielle du noeud.

Renvoie *copy* – copie superficielle du noeud.

Type renvoyé *Observation*

get_child_by_attribute (*attr*)

Retourne l'enfant du noeud qui correspond à l'attribut passé en argument, c'est-à-dire l'enfant sur branche `_yes_child` si « yes » est passé en argument et l'enfant sur branche `_no_child` if « no » est passé.

Paramètres *attr* (*str*) – Indique la branche voulue.

Renvoie *child* – L'enfant correspondant à l'attribut.

Type renvoyé *NodeST*

get_list_of_children ()

Retourne la liste avec tous les enfants du noeud.

Renvoie *list_of_children* – Liste avec tous les enfants du noeud.

Type renvoyé list(*NodeST*)

get_no_child ()

Getter de l'attribut `_no_child`.

Renvoie *_no_child* – Enfant du noeud qui correspond à la branche « no ».

Type renvoyé *NodeST*

get_obs_rep_couples ()

Getter de l'attribut `_obs_rep_couples`.

Renvoie *_obs_rep_couples* – Indique si le noeud représente un couple d'observation-réparation ou pas.

Type renvoyé bool

get_yes_child ()

Getter de l'attribut `_yes_child`.

Renvoie *_yes_child* – Enfant du noeud qui correspond à la branche « yes ».

Type renvoyé *NodeST*

set_child_by_attribute (*attr*, *child=None*)

Met en place l'enfant correspondant à l'attribut de la branche indiqué par *attr*.

Paramètres

— *attr* (*str*) – Indique la branche voulue.

— *child* (*NodeST*, *facultatif*) – Enfant qui va être mis en place.

set_no_child (*no_child=None*)

Setter de l'attribut `_no_child`.

Paramètres *no_child* (*NodeST*, *facultatif*) – Enfant du noeud qui correspond à la branche « no ».

set_obs_rep_couples (*obs_rep_couples*)

Setter de l'attribut *_obs_rep_couples*.

Paramètres *obs_rep_couples* (*bool*) – Indique si le noeud représente un couple d'observation-réparation ou pas.

set_yes_child (*yes_child=None*)

Setter de l'attribut *_yes_child*.

Paramètres *yes_child* (*NodeST*, *facultatif*) – Enfant du noeud qui correspond à la branche « yes ».

class StrategyTree.**Repair** (*id*, *cost*, *name=None*, *child=None*)

Classe pour représenter les noeuds des arbres de stratégie correspondants à des actions de réparation.

Paramètres

— **id** (*str*) – Identificateur unique du noeud.

— **cost** (*float*) – Correspond au « coût » du noeud.

— **name** (*str*, *facultatif*) – Nom du noeud qui peut ne pas être unique; si rien a été soumis, on pose que *_name = _id*.

— **child** (*NodeST*, *facultatif*) – Enfant du noeud, c'est-à-dire, le noeud suivant dans un arbre.

_id

Identificateur unique du noeud.

Type *str*

_cost

Correspond au « coût » du noeud.

Type *float*

name

Nom du noeud qui peut ne pas être unique; si rien a été soumis, on pose que *_name = _id*.

Type *str*, facultatif

_child

Enfant du noeud, c'est-à-dire, le noeud suivant dans un arbre.

Type *NodeST*

add_child (*child*)

Ajoute un enfant dans la liste des enfants du noeud. ATTENTION : ce méthode ne change pas un enfant qui existe déjà; pour cela, veuillez utiliser *set_child*.

Paramètres *child* (*NodeST*) – L'enfant à ajouter.

bn_labels_children_association ()

Retourne un dictionnaire des associations entre les labels d'un réseau Bayésien et les enfants du noeud.

Renvoie *da* – Dictionnaire des associations concerné.

Type renvoyé *dict*

copy ()

Retourne une copie superficielle du noeud.

Renvoie *copy* – Copie superficielle du noeud.

Type renvoyé *Repair*

get_child ()

Getter de l'attribut *_child*.

Renvoie *_child* – L'enfant du noeud concerné.

Type renvoyé *NodeST*

get_child_by_attribute (*attr*)

Realisation d'une méthode abstraite de la superclass; comme ce type de noeud ne dispose que d'un seul enfant on retourne toujours cet enfant pour n'importe quel *attr* passé en argument.

Paramètres *attr* (*str*) – L’attribut de l’enfant qu’il faut retourner. Peut être n’importe quoi ici.

Renvoie *child* – L’enfant du noeud concerné.

Type renvoyé *NodeST*

get_list_of_children ()

Retourne la liste qui contient tous les enfants du noeud ; pour ce cas, soit une liste avec un seul élément, soit une liste vide.

Renvoie *list_of_children* – La liste de tous les enfants du noeud (ici soit une liste avec un seul élément, soit une liste vide).

Type renvoyé *list(NodeST)*

set_child (*child=None*)

Setter de l’attribut *_child*.

Paramètres *child* (*NodeST*, *facultatif*) – L’enfant du noeud.

set_child_by_attribute (*attr*, *child=None*)

Realisation d’une méthode abstraite de la superclass qui met en place un enfant correspondant à *attr* au noeud concerné. superclass.

Paramètres

— *attr* (*str*) – L’attribut (un type) de l’enfant qu’il faut mettre en place.

— *child* (*NodeST*, *facultatif*) – L’enfant qu’on veut ajouter. Il faut qu’il corresponde à l’attribut soumis.

class *StrategyTree.StrategyTree* (*root=None*, *nodes=None*)

Représente l’arbre de stratégie qui est utilisé à la résolution du problème de Troubleshooting.

Paramètres

— *root* (*NodeST*, *facultatif*) – Racine de l’arbre, i.e. une action pour commencer.

— *nodes* (*list(NodeST)*, *facultatif*) – Liste des noeuds de l’arbre.

_root

Racine de l’arbre, i.e. une action pour commencer.

Type *NodeST*

_nodes

Liste des noeuds de l’arbre.

Type *list(NodeST)*

_adj_dict

Dictionnaire qui indique quels noeuds sont liés par des arcs.

Type *dict*

fout_newline

Indique le début d’une nouvelle ligne quand on transforme cet arbre en un fichier de texte.

Type *str*

fout_sep

Séparateur d’attributs qu’on utilise quand on transforme cet arbre en un fichier texte.

Type *str*

__str__ ()

Réalise la transformation de l’arbre vers *str*.

Renvoie *st_str* – Représentation de l’arbre de stratégie en forme de *str*.

Type renvoyé *str*

add_edge (*parent*, *child*, *child_type=None*)

Permet d’ajouter un arc dans un arbre entre deux noeuds.

Paramètres

- **parent** (*str* / *NodeST*) – Noeud qui va être le parent. Le noeud duquel l’arc part.
- **child** (*str* / *NodeST*) – Noeud qui va être l’enfant. Le noeud auquel l’arc arrive.
- **child_type** (*str*, *facultatif*) – L’attribut de la branche du parent à laquelle il faut ajouter l’enfant (par exemple si parent est une observation alors child_type est égal soit à “no”, soit à “yes”).

add_node (*node*)

Permet d’ajouter un ou plusieurs nouveaux noeuds dans l’arbre.

Paramètres **node** (*NodeST* / *list* (*NodeST*)) – Noeud.s à ajouter.

connect (*root_with_subtree*, *root_child_type=None*)

Connecte deux arbres, plus précisément, on connecte l’arbre actuel à l’arbre *root_with_subtree* en remplissant la branche qui correspond à *root_child_type* dans *root_with_subtree*.

Paramètres

- **root_with_subtree** (*StrategyTree*) – L’arbre vers la racine duquel on va connecter l’arbre actuel.
- **root_child_type** (*str*) – L’attribut de la branche de la racine du *root_with_subtree*.

Renvoie **merged_tree** – L’arbre fusionné.

Type renvoyé *StrategyTree*

copy ()

Retourne une copie superficielle de l’arbre.

Renvoie **copy** – Copie superficielle de l’arbre.

Type renvoyé *StrategyTree*

get_adj_dict ()

Getter de l’attribut *_adj_dict*.

Renvoie **_adj_dict** – Copie superficielle du dictionnaire d’adjacence de l’arbre.

Type renvoyé dict

get_edges ()

Récupère tous les arcs de l’arbre.

Renvoie **edges** – Liste de triplets où chaque élément correspond à un arc d’un graphe de manière que *tuple[0]* est un parent, *tuple[1]* est leur enfant et *tuple[2]* est l’attribut identifiant la branche.

Type renvoyé *list(tuple(NodeST, NodeST, str))*

get_node (*id*)

Retourne le noeud exacte (en sens de l’objet dans mémoire vivant) de l’arbre avec *id* indiqué.

Paramètres **id** (*str* / *NodeST*) – Soit *id* du noeud, soit un noeud lui-même dont on cherche un clone (en sens d’id) dans l’arbre.

Renvoie **n** – Noeud de l’arbre avec la *id* soumis.

Type renvoyé *NodeST*

get_node_by_name (*name*)

Retourne tous les noeuds de l’arbre dont les noms sont égaux à celui indiqué.

Paramètres **name** (*str*) – Un nom ou un noeud dont le nom on doit utiliser.

Renvoie **nodes** – Liste de tous les noeuds de l’arbre qui ont le même nom que celui indiqué.

Type renvoyé *list(NodeST)*

get_nodes ()

Getter de l’attribut *_nodes*.

Renvoie **_nodes** – Copie superficielle de la liste des noeuds de l’arbre.

Type renvoyé *list(NodeST)*

get_parent (*child*)

Retourne le parent du noeud *child* dans l'arbre. Remarque : dans cette implémentation d'arbre chaque noeud ne peut avoir qu'un seul parent.

Paramètres *child* (*str* / *NodeST*) – L'enfant dont le parent on cherche dans l'arbre.

Renvoie *parent* – Parent du noeud *child*.

Type renvoyé *NodeST*

get_root ()

Getter de l'attribut *_root*.

Renvoie *_root* – Racine de l'arbre.

Type renvoyé *NodeST*

get_sub_tree (*sub_root*)

Retourne le sous-arbre qui a le noeud *sub_root* comme racine.

Paramètres *sub_root* (*NodeST*) – Racine de le sous-arbre.

Renvoie *sub_tree* – Sous-arbre de cet arbre dont la racine est *sub_root*.

Type renvoyé *StrategyTree*

remove_sub_tree (*sub_root*)

Supprime le sous-arbre qui a comme racine *sub_root*.

Paramètres *sub_root* (*str* / *NodeST*) – Racine du sous-arbre qu'il faut supprimer.

Renvoie *flag* – Égale à True si la fonction a supprimé un sous-arbre, False sinon.

Type renvoyé bool

set_root (*root*)

Setter de l'attribut *_root*.

Paramètres *root* (*NodeST*) – Racine à mettre en place.

str_alt ()

Réalise la transformation de l'arbre vers *str*.

Renvoie *st_str* – Représentation de l'arbre de stratégie en forme de *str*.

Type renvoyé *str*

str_alt_2 ()

Transformation alternative de l'arbre vers *str*.

Renvoie *st_str* – Représentation de l'arbre de stratégie en forme de *str* de manière alternative.

Type renvoyé *str*

to_file (*filename*='last_best_tree.txt')

Permet de sauvegarder l'arbre de stratégie sous forme de fichier texte. On utilise le modèle suivant : 1) Chaque noeud est représenté par une ligne du type : *_id,_cost,_name,_type* C'est bien possible de remplacer la virgule par un séparateur différent en précisant l'attribut *self.fout_sep* de la classe. 2) Chaque arc est représenté par une ligne du type : *_id_parent,_id_child,_attribut* Où *_attribut* est le type d'arc (par exemple "yes" ou "no" si parent est une Observation). 3) Le fichier lui-même a la structure suivante : racine de l'arbre # ligne 1 [ligne vide] # ligne 2 noeud_1 # ligne 3 noeud_2 # ligne 4 ... noeud_n # ligne n + 2 [ligne vide] # ligne n + 3 arc_1 # ligne n + 4 arc_2 # ligne n + 5 ... arc_m # ligne n + m + 3 Cette méthode utilise également l'attribut *self.fout_newline* pour représenter le signe qui indique le début d'une nouvelle ligne.

Paramètres *filename* (*str*, *facultatif*) – Le nom du fichier où on sauvegarde le texte.

visualize (*filename*='last_best_strategy_tree.gv')

Affiche l'arbre de stratégie via le module *graphviz*. L'image construit est sauvegardé dans le fichier *filename.pdf*.

Paramètres *filename* (*str*, *facultatif*) – Le nom du fichier où on sauvegarde l'image.

`StrategyTree.st_from_file(filename='last_best_tree.txt', sep=',', newline=None)`

Permet de créer un objet du type `StrategyTree` à partir du fichier indiqué par *filename* en suivant le modèle fourni par la méthode `StrategyTree.to_file`.

Paramètres

- **filename** (*str*, facultatif) – Nom du fichier où l’arbre est stocké l’arbre sous forme textuelle.
- **sep** (*str*, facultatif) – Le séparateur utilisé dans le fichier.
- **newline** (*str*, facultatif) – Signe qui indique le début d’une nouvelle ligne.

Renvoie *stin* – L’arbre créé à partir des paramètres passés.

Type renvoyé *StrategyTree*

d

`DecisionTheoreticTroubleshooting`, [1](#)

s

`StrategyTree`, [10](#)

- `__eq__()`méthode StrategyTree.NodeST, 11
- `__str__()`méthode StrategyTree.NodeST, 11
- `__str__()`méthode StrategyTree.StrategyTree, 15
- `_adj_dictattribut` StrategyTree.StrategyTree, 15
- `_childattribut` StrategyTree.Repair, 14
- `_compute_costs()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 2
- `_costattribut` StrategyTree.NodeST, 10
- `_costattribut` StrategyTree.Observation, 12
- `_costattribut` StrategyTree.Repair, 14
- `_create_nodes()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 2
- `_evaluate_all_st()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 2
- `_expected_cost_of_repair_internal()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 3
- `_idattribut` StrategyTree.NodeST, 10
- `_idattribut` StrategyTree.Observation, 12
- `_idattribut` StrategyTree.Repair, 14
- `_nameattribut` StrategyTree.NodeST, 11
- `_nameattribut` StrategyTree.Observation, 12
- `_next_node_id()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 3
- `_no_childattribut` StrategyTree.Observation, 12
- `_nodesattribut` StrategyTree.StrategyTree, 15
- `_obs_rep_couplesattribut` StrategyTree.Observation, 12
- `_rootattribut` StrategyTree.StrategyTree, 15
- `_start_bay_lp()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 3
- `_yes_childattribut` StrategyTree.Observation, 12
- `add_child()`méthode StrategyTree.NodeST, 11
- `add_child()`méthode StrategyTree.Observation, 13
- `add_child()`méthode StrategyTree.Repair, 14
- `add_edge()`méthode StrategyTree.StrategyTree, 15
- `add_evidence()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 3
- `add_node()`méthode StrategyTree.StrategyTree, 16
- `bay_lpattribut` DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- `bayesian_networkattribut` DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- `bcolorsclasse` dans DecisionTheoreticTroubleshooting, 10
- `best_EVOI()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 4
- `bn_labels_children_association()`méthode StrategyTree.NodeST, 11
- `bn_labels_children_association()`méthode StrategyTree.Observation, 13
- `bn_labels_children_association()`méthode StrategyTree.Repair, 14
- `brute_force_solver()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 4
- `brute_force_solver_actions_only()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 4
- `brute_force_solver_tester()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 4
- `compute_EVOIs()`méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 5
- `connect()`méthode StrategyTree.StrategyTree, 16
- `copy()`méthode StrategyTree.NodeST, 11
- `copy()`méthode StrategyTree.Observation, 13
- `copy()`méthode StrategyTree.Repair, 14
- `copy()`méthode StrategyTree.StrategyTree, 16
- `costs_obsattribut` DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- `costs_repattribut` DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- `costs_rep_intervalattribut` DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- DecisionTheoreticTroubleshootingmodule, 1
- `diff_dicts()`dans le module DecisionTheoreticTroubleshooting, 10

- draw_true_prices()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 5
- dynamic_programming_solver()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 5
- ECR_ECO_wrapper()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 2
- elicitation()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 6
- elicitation_solver_tester()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 6
- evidencesattribut DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- expected_cost_of_repair()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 6
- expected_cost_of_repair_seq_of_actions()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 6
- fout_newlineattribut StrategyTree.StrategyTree, 15
- fout_sepattribut StrategyTree.StrategyTree, 15
- get_adj_dict()méthode StrategyTree.StrategyTree, 16
- get_child()méthode StrategyTree.Repair, 14
- get_child_by_attribute()méthode StrategyTree.NodeST, 11
- get_child_by_attribute()méthode StrategyTree.Observation, 13
- get_child_by_attribute()méthode StrategyTree.Repair, 14
- get_cost()méthode StrategyTree.NodeST, 11
- get_edges()méthode StrategyTree.StrategyTree, 16
- get_id()méthode StrategyTree.NodeST, 11
- get_list_of_children()méthode StrategyTree.NodeST, 11
- get_list_of_children()méthode StrategyTree.Observation, 13
- get_list_of_children()méthode StrategyTree.Repair, 15
- get_name()méthode StrategyTree.NodeST, 11
- get_no_child()méthode StrategyTree.Observation, 13
- get_node()méthode StrategyTree.StrategyTree, 16
- get_node_by_name()méthode StrategyTree.StrategyTree, 16
- get_nodes()méthode StrategyTree.StrategyTree, 16
- get_obs_rep_couples()méthode StrategyTree.Observation, 13
- get_parent()méthode StrategyTree.StrategyTree, 16
- get_proba()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 6
- get_root()méthode StrategyTree.StrategyTree, 17
- get_sub_tree()méthode StrategyTree.StrategyTree, 17
- get_yes_child()méthode StrategyTree.Observation, 13
- merge_dicts()dans le module DecisionTheoreticTroubleshooting, 10
- myopic_solver()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 7
- myopic_solver_st()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 7
- myopic_solver_tester()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 7
- myopic_wrapper()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 8
- nameattribut StrategyTree.Repair, 14
- NodeSTclasse dans StrategyTree, 10
- noeud_ant()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 8
- Observationclasse dans StrategyTree, 12
- observation_nodesattribut DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- observation_obsolete()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 8
- problem_defining_nodeattribut DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- remove_evidence()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 8
- remove_sub_tree()méthode StrategyTree.StrategyTree, 17
- Repairclasse dans StrategyTree, 14
- repairable_nodesattribut DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- reset_bay_lp()méthode DecisionTheoreticTroubleshooting.TroubleShootingProblem, 8
- service_nodeattribut DecisionTheoreticTroubleshooting.TroubleShootingProblem, 1
- set_child()méthode StrategyTree.Repair, 15
- set_child_by_attribute()méthode StrategyTree.NodeST, 12
- set_child_by_attribute()méthode StrategyTree.Observation, 13
- set_child_by_attribute()méthode StrategyTree.Repair, 15
- set_cost()méthode StrategyTree.NodeST, 12
- set_id()méthode StrategyTree.NodeST, 12
- set_name()méthode StrategyTree.NodeST, 12
- set_no_child()méthode StrategyTree.Observation, 13
- set_obs_rep_couples()méthode StrategyTree.Observation, 13
- set_root()méthode StrategyTree.StrategyTree, 17
- set_yes_child()méthode StrategyTree.Observation, 14
- shallow_copy_list_of_copyable()dans le module DecisionTheoreticTroubleshooting, 10
- shallow_copy_parent()dans le module DecisionTheoreticTroubleshooting, 10

`simple_solver()`méthode `DecisionTheoreticTroubleshooting.TroubleShootingProblem`, 8

`simple_solver_obs()`méthode `DecisionTheoreticTroubleshooting.TroubleShootingProblem`, 8

`simple_solver_obs_tester()`méthode `DecisionTheoreticTroubleshooting.TroubleShootingProblem`, 9

`simple_solver_tester()`méthode `DecisionTheoreticTroubleshooting.TroubleShootingProblem`, 9

`st_from_file()`dans le module `StrategyTree`, 17

`str_alt()`méthode `StrategyTree.StrategyTree`, 17

`str_alt_2()`méthode `StrategyTree.StrategyTree`, 17

`StrategyTree`classe dans `StrategyTree`, 15

`StrategyTree`module, 10

`to_file()`méthode `StrategyTree.StrategyTree`, 17

`TroubleShootingProblem`classe dans `DecisionTheoreticTroubleshooting`, 1

`unrepairable_nodes`attribut `DecisionTheoreticTroubleshooting.TroubleShootingProblem`, 1

`visualize()`méthode `StrategyTree.StrategyTree`, 17